SEQUENT

DYNIX/ptx Programming Guide

1003-48613-00

SEQUENT

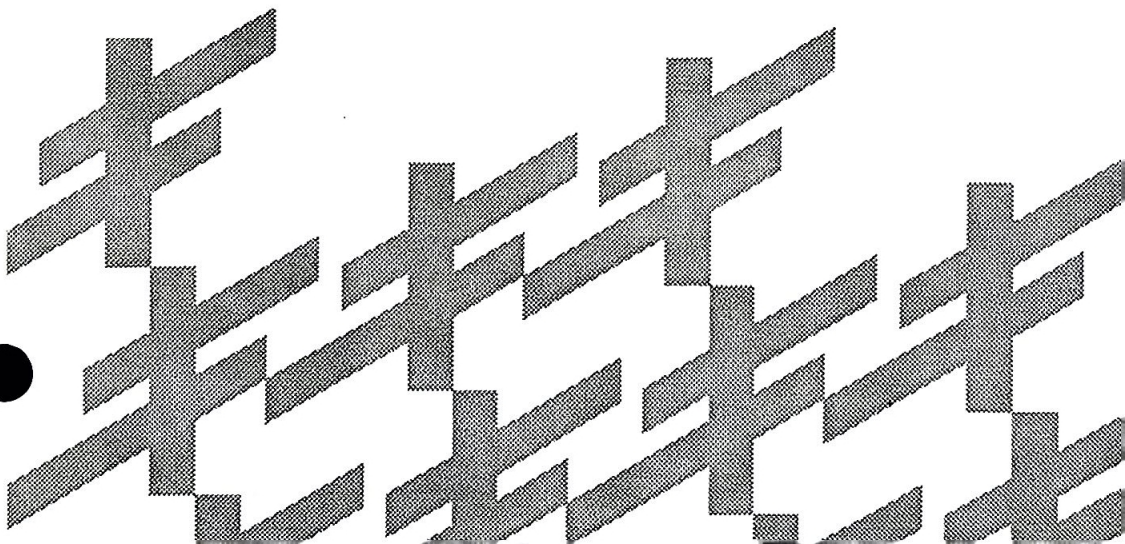
THIS SHEET HEADER
MUST BE PLACED ON
TOP OF CONTENTS

GRAPHIC PRODUCTS

Chicago, Illinois 60631

DYNIX/ptx®
Programming Guide

1003-48613-00
Sequent Computer Systems, Inc.



DYNIX and DYNIX/ptx are registered trademarks of Sequent Computer Systems, Inc.

Symmetry and Pdbx are trademarks of Sequent Computer Systems, Inc.

UNIX and Documenter's Workbench are registered trademarks of AT&T.

MULTIBUS is a trademark of Intel Corporation.

Ethernet is a trademark of Xerox Corporation.

VAX is a trademark of Digital Equipment Corporation.

AppleTalk is a trademark of Apple Computer Corporation.

Sun, Sun Microsystems, and NFS are trademarks of Sun Microsystems, Inc.

Copyright © 1985, 1986, 1987, 1988, 1989 by Sequent Computer Systems, Inc. All rights reserv
document may not be copied or reproduced in any form without permission from Sequent Compu
tems, Inc. Information in this document is subject to change without notice.

Printed in the United States of America.

Contents

About This Guide

Chapter 1. Programming Basics

1.1 Introduction 1-1
1.2 Choosing a Programming Language 1-1
1.3 After Your Code Is Written 1-6
1.4 The Programming Language–Operating System Interface 1-10
1.5 Analysis/Debugging 1-40
1.6 Program Organizing Utilities 1-56

Chapter 2. Application Programming Notes

2.1 Introduction 2-1
2.2 Application Programming 2-2
2.3 Language Selection 2-3
2.4 Advanced Programming Tools 2-10
2.5 Programming Support Tools 2-17
2.6 Project Control Tools 2-30
2.7 Example — **liber**, A Library System 2-33

Chapter 3. File and Record Locking

3.1 Introduction 3-1
3.2 Terminology 3-1
3.3 File Protection 3-3
3.4 Selecting Advisory or Mandatory Locking 3-14

Chapter 4. Interprocess Communication

4.1 Introduction 4-1
4.2 Messages 4-4
4.3 Semaphores 4-29
4.4 Shared Memory 4-57

Chapter 5. curses/terminfo

5.1 Introduction 5-1
5.2 Overview 5-2
5.3 Working with curses Routines 5-9
5.4 Working with terminfo Routines 5-70
5.5 Working with the terminfo Database 5-75
5.6 curses Program Examples 5-86

Chapter 6. Common Object File Format (COFF)

6.1 Introduction 6-1
6.2 Definitions and Conventions 6-3
6.3 File Header 6-4
6.4 Optional Header Information 6-6
6.5 Section Headers 6-9
6.6 Sections 6-13
6.7 Relocation Information 6-13
6.8 Line Numbers 6-15
6.9 Symbol Table 6-16
6.10 String Table 6-41
6.11 Access Routines 6-42

Figures

1-1 Manual Page for **gets**(3S) 1-22
1-2 A version of stdio.h 1-25

5-1 The stdscr/terminal screen relationship 5-16
5-2 Multiple windows and pads 5-18
5-3 The window/terminal screen relationship 5-59

6-1 Object File Format 6-2
6-2 COFF Symbol Table 6-17
6-3 Special Symbols (**.bb** and **.eb**) 6-19
6-4 Example of the Symbol Table 6-20
6-5 Symbols for Functions 6-21
6-5 String Table 6-41

Tables

1-1 C Language Standard I/O Subroutines 1-15
1-2 String Operations 1-17
1-3 Classifying ASCII Character-Coded Integer Values 1-19
1-4 Conversion Functions and Macros 1-20
1-5 Environment and Status System Calls 1-32
1-6 Process Status 1-33
1-7 prof Output 1-54
1-8 nm Output, with –f Option 1-59

2-1 Object File Interface Macros 2-21
2-2 Trigonometric Functions 2-25

2-3 Hyperbolic Functions 2-26
2-4 Miscellaneous Functions 2-27

4-1 Message Operation Permission Codes 4-9
4-2 Control Commands (Flags) 4-10
4-3 Operation Permissions Codes 4-36
4-4 **semget** Control Commands (Flags) 4-36
4-5 Shared Memory State Information 4-60
4-6 shmflg Operation Permissions Codes 4-63
4-7 shmget Control Commands (Flags) 4-64

5-1 Components of the curses/terminfo Screen Management System 5-7
5-2 The Default Colors Table 5-42
5-3 Example of a Pairs Table 5-43
5-4 Input Option Settings for curses Programs 5-52

6-1 File Header Contents 6-4
6-2 File Header Flags 6-5
6-3 Optional Header Contents 6-8
6-4 System Magic Numbers 6-8
6-5 Section Header Contents 6-10
6-6 Section Header Flags 6-11
6-7 Relocation Section Contents 6-13
6-8 Relocation Types 6-14
6-9 Line Number Grouping 6-15
6-10 Special Symbols in the Symbol Table 6-18
6-11 Symbol Table Entry Format 6-22
6-12 Name Field 6-23
6-13 Storage Classes 6-24
6-14 Storage Class by Special Symbols 6-25
6-15 Restricted Storage Classes 6-26
6-16 Storage Class and Value 6-27
6-17 Section Number 6-28
6-18 Section Number and Storage Class 6-29
6-19 Fundamental Types 6-30
6-20 Derived Types 6-31
6-21 Type Entries by Storage Class 6-32
6-22 Auxiliary Symbol Table Entries 6-34
6-23 Format for Auxiliary Table Entries for Sections 6-35
6-24 Tag Names Table Entries 6-36
6-25 Table Entries for End of Structures 6-36
6-26 Table Entries for Functions 6-37

6-27 Table Entries for Arrays 6-37
6-28 End of Block and Function Entries 6-38
6-29 Format for Beginning of Block and Function 6-38
6-30 Entries for Structures, Unions, and Enumerations 6-39

About This Guide

Overview

This guide describes programming methodology as applied in the DYNIX/ptx operating system. Each chapter offers a broad scope of information about concepts necessary to successful programming.

Chapters at a Glance

Chapter 1—Programming Basics
: Contains information designed to give the reader a good understanding of the programming basics needed to produce successful code in the operating system environment.

Chapter 2—Application Programming
: Contains information about application programming.

Chapter 3—File and Record Locking
: Introduces basic concepts about file and record locking and offers advice and examples of proper protection.

Chapter 4—Interprocess Communication
: Contains an exhaustive study of interprocess communication with many examples.

Chapter 5—curses/terminfo
: Contains information about screen and terminal I/O.

Chapter 6—Common Object File Format (COFF)
: Introduces and explains the form and function of object files in the programming environment.

Assumptions About the Reader

This guide is intended for programmers working in the DYNIX/ptx environment. It assumes the reader has a solid understanding of the operating system and knowledge of programming and computer operations.

Notational Conventions

The following notational conventions are used in this manual:

- Filenames and user-defined parameters in text are shown in *italics*.
- System output is shown in `even-width` font.
- Commands in text and examples of input to be entered literally are shown in **bold**.
- Environment variables are shown in **bold** upper or lowercase.

Chapter 1
Programming Basics

1.1 Introduction 1-1

1.2 Choosing a Programming Language 1-1
Supported Languages in the System Environment 1-2
C Language 1-2
FORTRAN 1-3
Pascal 1-3
COBOL 1-4
Assembly Language 1-4
Special-Purpose Languages 1-4
awk 1-4
lex 1-5
yacc 1-5
M4 1-6
bc and dc 1-6
curses 1-6

1.3 After Your Code Is Written 1-6
Compiling and Link Editing 1-7
Compiling C Programs 1-7
Compiling FORTRAN Programs 1-8
Compiler Diagnostic Messages 1-8
Link Editing 1-8

1.4 The Programming Language–Operating System Interface 1-10
Why C Is Used to Illustrate the Interface 1-10
How Arguments Are Passed to a Program 1-11
System Calls and Subroutines 1-13
Categories of System Calls and Subroutines 1-14
Where the Manual Pages Can Be Found 1-21
How System Calls and Subroutines Are Used in C Programs 1-21
Header Files and Libraries 1-27
Object File Libraries 1-28
Input/Output 1-29
Three Files You Always Have 1-29
Named Files 1-30

Low-Level I/O and Why You Shouldn't Use It 1-31
System Calls for Environment or Status Information 1-32
Processes 1-33
system(3S) 1-34
exec(2) 1-34
fork(2) 1-35
Pipes 1-37
Error Handling 1-38
Signals and Interrupts 1-38

1.5 Analysis/Debugging 1-40
Sample Program 1-40
cflow 1-43
ctrace 1-45
cxref 1-48
lint 1-52
prof 1-53
size 1-55
strip 1-55
Pdbx 1-56

1.6 Program Organizing Utilities 1-56
The make Command 1-56
The Archive 1-58
Use of SCCS by Single-User Programmers 1-62

Figures

1-1 Manual Page for **gets**(3S) 1-22
1-2 A version of stdio.h 1-25

Tables

1-1 C Language Standard I/O Subroutines 1-15
1-2 String Operations 1-17
1-3 Classifying ASCII Character-Coded Integer Values 1-19
1-4 Conversion Functions and Macros 1-20
1-5 Environment and Status System Calls 1-32
1-6 Process Status 1-33
1-7 prof Output 1-54
1-8 nm Output, with –f Option 1-59

Chapter 1
Programming Basics

1.1 Introduction

The information in this chapter is for anyone learning to write programs to run in a DYNIX/ptx system environment. People in the "single-user programmer" category, particularly those who are not deeply interested in programming, may find that this chapter (plus related man pages) tells them as much as they need to know about coding and running programs on a system computer.

Programmers whose interest runs deeper, who are part of an application development project, or who are producing programs on one system computer that are being ported to another, should view this chapter as a starter package.

1.2 Choosing a Programming Language

How do you decide which programming language to use in a given situation? One answer could be, "I always code in HAIRBOL, because that's the language I know best." In some circumstances that's a legitimate answer. But assuming that more than one programming language is available to you, that different programming languages have their strengths and weaknesses, and assuming that once you've learned to use one programming language it becomes relatively easy to learn to use another, you might approach the problem of language selection by asking yourself questions like the following:

- What is the nature of the task this program is to do? Does the task call for the development of a complex algorithm, or is this a simple procedure that has to be done on a lot of records?
- Does the programming task have many separate parts? Can the program be divided into separately compilable functions, or is it one module?
- How soon does the program have to be available? Is it needed right now, or do I have enough time to work out the most efficient process possible?

- What is the scope of its use? Am I the only person who will use this program, or is it going to be distributed to the whole world?
- Is there a possibility the program will be ported to other systems?
- What is the life-expectancy of the program? Is it going to be used just a few times, or will it still be going strong five years from now?

1.2.1 Supported Languages in the System Environment

By "supported languages" we mean those offered by for use on a your computer. Since these are separately purchasable items, not all of them will necessarily be installed on your machine. On the other hand, you may have languages available on your machine that came from another source and are not mentioned in this discussion. Be that as it may, in this section and the one to follow we give brief descriptions of the nature of six full-scale programming languages, and a number of special-purpose languages.

C Language

The C language is intimately associated with the operating system. If you need to use a lot of system function calls for low-level I/O, memory or device management, or interprocess communication, C language is a logical first choice. Most programs, however, don't require such direct interfaces with the operating system, so the decision to choose C might better be based on one or more of the following characteristics:

- A variety of data types: character, integer, long integer, float, and double
- Low level constructs (most of the system kernel is written in C)
- Derived data types such as arrays, functions, pointers, structures, and unions
- Multidimensional arrays
- Scaled pointers, and the ability to do pointer arithmetic
- Bit-wise operators
- A variety of flow-of-control statements: if, if-else, switch, while, do-while, and for
- A high degree of portability

C is a language that lends itself readily to structured programming. It is natural, in C, to think in terms of functions. The next logical step is to view each function as a separately compilable unit. This approach (coding a program in small pieces) eases the job of making changes and improvements. If this begins to sound like the UNIX system philosophy of building new programs from existing tools, it's not just coincidence. As you create functions for one program, you will surely find that many can be picked up, or quickly revised, for another program.

A difficulty with C is that it takes a fairly concentrated use of the language over a period of several months to reach your full potential as a C programmer. If you are a casual programmer, you might be better off choosing a less-demanding language.

FORTRAN

The oldest of the high-level programming languages, FORTRAN is still valued highly for its variety of mathematical functions. If you are writing a program for statistical analysis or other scientific applications, FORTRAN is a good choice. An original design objective was to produce a language with good operating efficiency. This has been achieved at the expense of some flexibility in the area of type definition and data abstraction. FORTRAN also requires using a somewhat rigid format for input of lines of source code. This shortcoming may be overcome by using one of the system tools designed to make FORTRAN more flexible.

Pascal

Originally designed as a teaching tool for block structured programming, Pascal has gained wide acceptance because of its straightforward style. Pascal is highly structured and allows system level calls (characteristics it shares with C). Since the intent of the developers, however, was to produce a language to teach people about programming, it is perhaps best suited to small projects. Among its inconveniences are its lack of facilities for specifying initial values for variables and limited file processing capability.

COBOL

Probably more programmers are familiar with COBOL than with any other single programming language. It is frequently used in business applications because its strengths lie in the management of input/output and in defining record layouts.

COBOL was not designed to handle complex algorithms, but it works well in cases where many records have to be passed through a simple process (a payroll withholding tax calculation, for example). The COBOL language is wordy, so the compilation process can be quite complex. Once written and put into production, COBOL programs have a way of staying in use for years, and what might at first be viewed as wordiness comes to be considered self-documentation.

Assembly Language

Assembly language, the closest thing to machine code, is specific to the computer on which your program is to run. High-level languages are translated into the assembly language for a specific processor as one step of the compilation. The most common reason for needing to work in assembly language arises when you want to do some task that is not within the scope of a high-level language. Since assembly language is machine-specific, programs written in it are not portable.

1.2.2 Special-Purpose Languages

In addition to the formal programming languages, the system environment frequently offers one or more of the special-purpose languages listed below.

In addition to what follows, don't overlook the possibility of using shell procedures.

awk

awk (an acronym constructed from the initials of its developers) scans an input file for lines that match patterns described in a specification file. On finding a line that matches a pattern, **awk** performs actions that are also described in the specification. An **awk** program that can be written in a couple of lines can often do functions that would take a couple of pages to describe in a programming language like FORTRAN or C. For example, consider a case where you have a set of records that consist of a key field and a second field that represents a quantity. You have sorted the records by the key field; you now want to add the quantities for records with duplicate keys and output a file in which no keys are duplicated. The pseudocode for such a program

might look like this:

```
Read the first record into a hold area;
Read additional records until EOF;
 {
 If the key matches the key of the record in the hold area,
  add the quantity to the quantity field of the held record;
 If the key does not match the key of the held record,
  write the held record,
  move the new record to the hold area;
 }
At EOF, write out the last record from the hold area.
```

An **awk** program to accomplish this task would look like this:

```
    { qty[$1] += $2 }
END { for (key in qty) print key, qty[key] }
```

This illustrates only one characteristic of **awk**: its ability to work with associative arrays. With **awk**, the input file does not have to be sorted, which is a requirement of the pseudoprogram.

lex

lex is a lexical analyzer that can be added to C or FORTRAN programs. A lexical analyzer is interested in the vocabulary of a language rather than its grammar, which is a system of rules defining the structure of a language. **lex** can produce C language subroutines that recognize regular expressions specified by the user, take some action when a regular expression is recognized, and pass the output stream on to the next program.

yacc

yacc (Yet Another Compiler Compiler) is a tool for describing an input language to a computer program. **yacc** produces a C language subroutine that parses an input stream according to rules laid down in a specification file. The **yacc** specification file establishes a set of grammar rules together with actions to be taken when tokens in the input match the rules. **lex** may be used with **yacc** to control the input process and pass tokens to the parser that applies the grammar rules.

M4

M4 is a macro processor that can be used as a preprocessor for any language. It is not, however, the preprocessor for C programs. It is described in Section (1) of the *Reference Manual.*

bc and dc

bc enables you to use a computer terminal as you would a programmable calculator. You can edit a file of mathematical computations and call **bc** to execute them. The **bc** program uses **dc**, which derives its name from "desk calculator". You can use **dc** directly, but it takes a little getting used to since it works with "reverse notation." That means you enter the operator after you enter all the numbers. **bc** and **dc** are described in Section (1) of the man pages.

curses

Actually a library of C functions, **curses** is included in this list because its set of functions resembles a sublanguage for dealing with terminal screens. If you are writing programs that include interactive user screens, you will want to become familiar with this group of functions.

1.3 After Your Code Is Written

Most compilation systems produce assembly language code as an intermediate step. The assembler, usually automatically, translates that code into the machine language of the computer the program is to run on. The link editor resolves all undefined references and makes the object module executable. With most languages in the operating system, the assembler and link editor produce files in what is known as the Common Object File Format (COFF). A common format makes it easier for utilities that depend on information in the object file to work on different machines running different versions of the operating system.

A COFF contains a header, relocation line numbers and symbol information, and two or more sections containing machine object code and data. Options of the compilers cause different items of information to be included in the COFF. For example, compiling a program with the **-g** option adds line numbers and other symbolic information. You can spend many years programming without having to worry too much about the contents and organization of the Common Object File Format, so we are not going into any further depth of detail at this point. Detailed information is available in Chapter 6 of this guide.

1.3.1 Compiling and Link Editing

The command used for compiling depends on the language used.

- For C programs, **cc** both compiles and link edits.
- For FORTRAN programs, **fortran** both compiles and link edits.

Compiling C Programs

To use the C compilation system, you must have your source code in a file with a filename that ends in the characters **.c**, as in **mycode.c**. The command to invoke the compiler is **cc** *mycode.c*. If the compilation is successful, the process proceeds through the link edit stage. The result will be an executable file with the name **a.out**.

Several options to the **cc** command are available to control its operation. The most used options are the following:

\nc	Causes the compilation system to suppress the link edit phase. This produces an object file (**mycode.o**) that can be link edited at a later time.
\ng	Causes the compilation system to generate special information about variables and language statements. If you are going through the stage of debugging your program, use this option.
\nO	Causes the inclusion of an additional optimization phase. This option is logically incompatible with the **\ng** option. You would normally use **\nO** after the program has been debugged, to reduce the size of the object file and increase execution speed.
\np	Causes the compilation system to produce code that works in conjunction with the **prof**(1) command to produce a runtime profile of where the program is spending its time. Useful in identifying which routines are candidates for improved code to reduce execution time.
–o *outfile*	Tells **cc** to tell the link editor to use the specified name for the executable file, rather than the default **a.out**.

Other options can be used with **cc**. Check the *Reference Manual*.

If you enter the **cc** command using a file name that ends in **.s**, the compilation system treats it as assembly language source code and bypasses all the steps ahead of the assembly step.

If you enter the **cc** command using a file that ends in **.o**, the compilation system treats it as COFF and bypasses all steps ahead of linking.

Compiling FORTRAN Programs

The **fortran** command invokes the FORTRAN compilation system. The operation of the command is similar to that of the **cc** command, except the source code file(s) must have a **.f** suffix. The **fortran** command compiles your source code and calls in the link editor to produce an executable file whose name is **a.out**.

The following command line options have the same meaning as they do for the **cc** command:

–c, –p, –O, –g, and **–o** *outfile*

Compiler Diagnostic Messages

The C compiler generates error messages for statements that don't compile. The messages are generally quite understandable, but—as with most language compilers—the messages sometimes point several statements beyond where the actual error occurred. For example, if you inadvertently put an extra semicolon (;) at the end of an **if** statement, a subsequent **else** will be flagged as a syntax error. In the case where a block of several statements follows the **if**, the line number of the syntax error caused by the **else** will start you looking for the error well past where it actually is. Unbalanced curly braces, { }, are another common cause of error messages far removed from the actual problem.

Link Editing

The **ld** command invokes the link editor directly. The typical user, however, seldom invokes **ld** directly. A more common practice is to use a language compilation control command (such as **cc**) that invokes **ld**. The link editor combines several object files into one, performs relocation, resolves external symbols, incorporates start-up routines, and supports symbol table information. You may, of course, start with a single object file rather than several. The resulting executable module is left in a file named **a.out**.

Any file named on the **ld** command line that is not an object file (typically, a name ending in **o**) is assumed to be an archive library or a file of link editor directives. The **ld** command has 16 options; this guide describes three of them (for more information see the *Programming Tools Guide*). These options should be fed to the link editor by specifying them on the **cc** command line if you are doing both jobs with the single command, which is the usual case.

–o *outfile* — Provides a name to be used to replace **a.out** as the name of the output file. Obviously, the name **a.out** is of only temporary usefulness. If you know the name you want use to invoke your program, you can provide it here. Of course, it may be equally convenient use the following command when you want to give your program a less temporary name:

```
mv a.out progname
```

–l*x* — Directs the link editor to include library routines from **lib***x***.a**, where *x* is up to nine characters. For C programs, **libc.a** is automatically searched if the **cc** command is used. The **–l***x* option is used to bring in libraries not normally in the search path such as **libm.a**, the math library. The **–l***x* option can occur more than once on a command line, with different values for the *x*. A library is searched when its name is encountered, so the placement of the option on the command line is important. The safest place to put it is at the end of the command line. The **–l***x* option is related to the **–L** option.

–L *dir* — Changes the **lib***x***.a** search sequence to search in the specified directory before looking in the default library directories, usually */lib* or */usr/lib*. This is useful if you have different versions of a library and you want to point the link editor to the correct one. It works on the assumption that once a library has been found no further searching for that library is necessary. Because **–L** diverts the search for the libraries specified by **–l***x* options, it must precede such options on the command line.

When the link editor is called through **cc**, a startup routine (typically */lib/crt0.o* for C programs) is linked with your program. This routine calls **exit**(2) after execution of the main program.

1.4 The Programming Language–Operating System Interface

When a program is run, it depends on the computer's operating system for a variety of services. Some of the services, such as bringing the program into main memory and starting the execution, are completely transparent to the program. They are, in effect, arranged for in advance by the link editor when it marks an object module as executable. As a programmer, you seldom need to be concerned about such matters.

Other services, however, such as input/output, file management, and storage allocation do require work on the part of the programmer. These connections between a program and the operating system are what is meant by the term system–language interface. The following topics are included in this section:

- How arguments are passed to a program
- System calls and subroutines
- Header files and libraries
- Input/Output
- Processes
- Error Handling, Signals, and Interrupts

1.4.1 Why C Is Used to Illustrate the Interface

Throughout this section, C programs are used to illustrate the interface between the operating system and programming languages because C programs make more use of the interface mechanisms than other high-level languages. What is really being covered in this section, then, is the operating system–C Language interface.

1.4.2 How Arguments Are Passed to a Program

Information or control data can be passed to a C program as arguments on the command line. When the program is run as a command, arguments on the command line are made available to the function **main** in two parameters, an argument count and an array of pointers to character strings. (Every C program is required to have an entry module named **main**.) Since the argument count is always given, the program does not have to know in advance how many arguments to expect. The character strings pointed at by elements of the array of pointers contain the argument information.

The arguments are presented to the program traditionally as **argc** and **argv**, although any names you choose will work. **argc** is an integer that gives the count of the number of arguments. Since the command itself is considered to be the first argument, **argv[0]**, the count is always at least one. **argv** is an array of pointers to character strings (character strings are arrays of characters terminated by the null character \0).

If you plan to pass runtime parameters to your program, you need to include code to deal with the information. Here are two possible uses of runtime parameters:

- As control data. Use the information to set internal flags that control the operation of the program.
- To provide a variable filename to the program.

The following two program fragments illustrate these uses.

Example 1

```
#include <stdio.h>

main(argc, argv)
  int argc;
  char *argv[];
{
    void exit();
    int oflag = FALSE;
    int pflag = FALSE;   /* Function Flags */
    int rflag = FALSE;
    int ch;

        while ((ch = getopt(argc,argv, "opr")) != EOF)
        {
          /* For options present, set flag to TRUE */
```

```
        /* If no options present, print error message */
          switch (ch)
          {
          case 'o':
              oflag = 1;
              break;
          case 'p':
              pflag = 1;
              break;
          case 'r':
              rflag = 1;
              break;
          default:
              (void)fprintf(stderr,
              "Usage: %s [-opr]\n", argv[0]);
              exit(2);
          }
      }
      .
      .
      .
}
```

Example 2

```
#include <stdio.h>

main(argc, argv)
  int argc;
  char *argv[];
{
    FILE *fopen(), *fin;
    void perror(), exit();

    if (argc > 1)
    {
      if ((fin = fopen(argv[1], "r")) == NULL)
      {
        /* First string (%s) is program name (argv[0]) */
        /* Second string (%s) is name of file that could */
        /* not be opened (argv[1]) */

        (void)fprintf(stderr,
          "%s: cannot open %s: ", argv[0], argv[1]);
```

```
            perror(argv[0]);
            exit(2);
          }
      }
      .
      .
      .
}
```

The shell, which makes arguments available to your program, considers an argument to be any non-blank characters separated by blanks or tabs. Characters enclosed in double quotes ("abc def") are passed to the program as one argument even if blanks or tabs are among the characters. It goes without saying that you are responsible for error checking and otherwise making sure that the argument received is what your program expects it to be.

In addition to **argc** and **argv**, a third argument is also present. The third argument, known as **envp**, is an array of pointers to environment variables. You can find more information on **envp** in the *Reference Manual* under **exec**(2) and **environ**(5).

1.4.3 System Calls and Subroutines

System calls are requests from a program for an action to be performed by the operating system kernel. Subroutines are precoded modules used to supplement the functionality of a programming language.

Both system calls and subroutines resemble functions you might code for the individual parts of your program. There are, however, differences between them:

- At link edit time, the code for subroutines is copied into the object file for your program; the code invoked by a system call remains in the kernel.
- At execution time, subroutine code is executed as if it was code you had written yourself; a system function call is executed by switching from your process area to the kernel.

This means that while subroutines make your executable object file larger, runtime overhead for context switching may be less and execution may be faster.

Categories of System Calls and Subroutines

System calls (see Section 2 in the *Reference Manual*) can be divided into the following categories:

- File access
- File and directory manipulation
- Process control
- Environment control and status information

You can generally tell the category of a subroutine by the section of the *Reference Manual* in which you find its manual page. However, the first part of Section 3 (3C and 3S) covers such a variety of subroutines that it might be helpful to classify them further.

- The subroutines of sub-class 3S constitute the operating system/C Language standard I/O, an efficient I/O buffering scheme for C.
- The subroutines of subclass 3C do a variety of tasks. They have in common the fact that their object code is stored in **libc.a**. They can be divided into the following categories:
    - String manipulation
    - Character conversion
    - Character classification
    - Environment management
    - Memory management

Table 1-1 lists the functions that compose the standard I/O subroutines. Frequently, one manual page describes several related functions. In Table 1-1, the left-hand column contains the name that appears at the top of the manual page; the other names in the same row are related functions described on the same manual page.

Table 1-1
C Language Standard I/O Subroutines

Function Name(s)	Purpose
fclose, fflush	Close or flush a stream
ferror, feof, clearerr, fileno	Stream status inquiries
fopen, freopen, fdopen	Open a stream
fread, fwrite	Binary input/output
fseek, rewind, ftell	Reposition a file pointer in a stream
getc, getchar, fgetc, getw	Get a character or word from a stream
gets, fgets	Get a string from a stream
popen, pclose	Begin or end a pipe to/from a process
printf, fprintf, sprint	Print formatted output
For all functions: #include <stdio.h>	

Table 1-1
C Language Standard I/O Subroutines (cont.)

Function Name(s)	Purpose
putc, putchar, fputc, putw	Put a character or word on a stream
puts, fputs	Put a string on a stream
scanf, fscanf, sscanf	Convert formatted input
setbuf, setvbuf	Assign buffering to a stream
system	Issue a command through the shell
tmpfile	Create a temporary file
tmpnam, tempnam	Create a name for a temporary file
ungetc	Push character back into input stream
vprintf, vfprintf, vsprintf	Print formatted output of a varargs argument list
For all functions: #include <stdio.h>	

Table 1-2 lists string handling functions that are grouped under the heading **string**(3C) in the *Reference Manual.*

Table 1-2
String Operations

Operator	Function
strcat(*s1, s2*)	Append a copy of *s2* to the end of *s1*.
strncat(*s1, s2, n*)	Append *n* characters from *s2* to the end of *s1*.
strcmp(*s1, s2*)	Compare two strings. Returns an integer less than, greater than or equal to 0 to show that *s1* is lexicographically less than, greater than or equal to *s2*.
strncmp(*s1, s2, n*)	Compare n characters from the two strings. Results are otherwise identical to **strcmp**.
strcpy(*s1, s2*)	Copy *s2* to *s1*, stopping after the null character (\0) has been copied.
strncpy(*s1, s2, n*)	Copy *n* characters from *s2* to *s1*. *s2* will be truncated if it is longer than *n*, or padded with null characters if it is shorter than *n*.
strdup(*s*)	Returns a pointer to a new string that is a duplicate of the string pointed to by *s*.
strchr(*s, c*)	Returns a pointer to the first occurrence of character *c* in string *s*, or a NULL pointer if *c* is not in *s*.
strrchr(*s, c*)	Returns a pointer to the last occurrence of character *c* in string *s*, or a NULL pointer if *c* is not in *s*.
For all functions: **#include <string.h>** **string.h** provides extern definitions of the string functions.	

Table 1-2
String Operations (cont.)

Operator	Function
strlen(*s*)	Returns the number of characters in *s* up to the first null character.
strpbrk(*s1, s2*)	Returns a pointer to the first occurrence in *s1* of any character from *s2*, or a NULL pointer if no character from *s2* occurs in *s1*.
strspn(*s1, s2*)	Returns the length of the initial segment of *s1*, which consists entirely of characters from *s2*.
strcspn(*s1, s2*)	Returns the length of the initial segment of *s1*, which consists entirely of characters not from *s2*.
strtok(*s1, s2*)	Look for occurrences of *s2* within *s1*.
For all functions: **#include <string.h>** **string.h** provides extern definitions of the string functions.	

Table 1-3 lists macros that classify ASCII character-coded integer values. These macros are described under the heading **ctype**(3C) in Section 3 of the *Reference Manual.*

Table 1-3
Classifying ASCII Character-Coded Integer Values

Operator	Function
isalpha(*c*)	Is *c* a letter (TRUE if *c* is a letter)?
isupper(*c*)	Is *c* an upper-case letter?
islower(*c*)	Is *c* a lower-case letter?
isdigit(*c*)	Is *c* a digit [0-9]?
isxdigit(*c*)	Is *c* a hexadecimal digit [0-9], [A-F] or [a-f]?
isalnum(*c*)	Is *c* an alphanumeric (letter or digit)?
isspace(*c*)	Is *c* a space, tab, carriage return, new-line, vertical tab or form-feed?
ispunct(*c*)	Is *c* a punctuation character (neither control nor alphanumeric)?
isprint(*c*)	Is *c* a printing character, code 040 (space) through 0176 (tilde)?
isgraph(*c*)	Same as isprint except false for 040 (space)
iscntrl(*c*)	Is *c* a control character (less than 040) or a delete character? (0177)
isascii(*c*)	Is *c* an ASCII character (code less than 0200)?
For all functions: **#include <ctype.h>** Nonzero return == true; zero return == false	

Table 1-4 lists functions and macros that are used to convert characters, integers, or strings from one representation to another.

Table 1-4
Conversion Functions and Macros

Function Name(s)			Purpose
a64l	**l64a**		Convert between long integer and base-64 ASCII string.
ecvt	**fcvt**	**gcvt**	Convert floating-point number to string.
l3tol	**ltol3**		Convert between 3-byte integer and long integer.
strtod	**atof**		Convert string to double-precision number.
strtol	**atol**	**atoi**	Convert string to integer.

conv(3C):	Translate Characters
toupper	Lowercase to uppercase.
_toupper	Macro version of **toupper**.
tolower	Upper-case to lower-case.
_tolower	Macro version of **tolower**.
toascii	Turn off all bits that are not part of a standard ASCII character; intended for compatibility with other systems.

For all **conv**(3C) macros: **#include <ctype.h>**

Where the Manual Pages Can Be Found

System calls are listed alphabetically in Section 2 of the *Reference Manual.* Subroutines are listed in Section 3. We have described above what is in the first subsection of Section 3, Section 3C. The remaining subsections of Section 3 are:

- 3F—the FORTRAN intrinsic function library, **libf**
- 3M—functions that make up the Math Library, **libm**
- 3N—Networking Support Utilities
- 3PPS—the parallel programming library
- 3S—the standard I/O package, **stdio**(3S)
- 3SEQ—functions specific to this operating system
- 3X—various specialized functions

How System Calls and Subroutines Are Used in C Programs

Information about the proper way to use system calls and subroutines is given on the manual page. To illustrate, a typical manual page (for **gets**(3S)) is shown in Figure 1-1.

NAME

gets, fgets - get a string from a stream

SYNOPSIS

```
#include <stdio.h>

char  *gets (s)
char  *s;

char  *fgets (s, n, stream)
char  *s;
int   n;
FILE  *stream;
```

DESCRIPTION

Gets reads characters from the standard input stream, *stdin*, into the array pointed to by *s*, until a new-line character is read or an end-of-file condition is encountered. The new-line character is discarded and the string is terminated with a null character.

Fgets reads characters from the *stream* into the array pointed to by *s*, until *n*-1 characters are read, or a new-line character is read and transferred to *s*, or an end-of-file condition is encountered. The string is then terminated with a null character.

SEE ALSO

ferror(3S)
fopen(3S)
fread(3S)
getc(3S)
scanf(3S)

DIAGNOSTICS

If end-of-file is encountered and no characters have been read, no characters are transferred to *s* and a NULL pointer is returned. If a read error occurs, such as trying to use these functions on a file that has not been opened for reading, a NULL pointer is returned. Otherwise *s* is returned.

Figure 1-1. Manual Page for gets(3S).

As you can see from the illustration, two related functions are described on this page: **gets** and **fgets**. Each function gets a string from a stream in a slightly different way. The DESCRIPTION section tells how each operates.

It is the SYNOPSIS section, however, that contains the critical information about how the function (or macro) is used in your program. Notice that the first line in the SYNOPSIS is:

#include <stdio.h>

This means that to use **gets** or **fgets** you must bring the standard I/O header file into your program (generally right at the top of the file). There is something in **stdio.h** that is needed when you use the described functions. A version of **stdio.h** is shown in Figure 1-2. Check it to see if you can understand what **gets** or **fgets** uses.

The next thing shown in the SYNOPSIS section of a manual page that documents system calls or subroutines is the formal declaration of the function. The formal declaration gives you the following information:

- **The type of object returned by the function.** In our example, both **gets** and **fgets** return a character pointer.
- **The object or objects the function expects to receive when called.** These are the items enclosed in the parentheses following the function. **gets** expects a character pointer. (The DESCRIPTION section explains what the tokens of the formal declaration stand for.)
- **How the function is going to treat those objects.** The declaration

  ```
  char *s;
  ```

  in **gets** means that the token **s** enclosed in the parentheses will be considered to be a pointer to a character string. Bear in mind that in the C language, when passed as an argument, the name of an array is converted to a pointer to the beginning of the array.

We have chosen a simple example here in **gets**. If you want to try something a little more complex, work out the meaning of the elements of the **fgets** declaration.

While we're on the subject of **fgets**, there is another piece of C esoterica. Notice that the third parameter in the **fgets** declaration is referred to as **stream**. A **stream**, in this context, is a file with its associated buffering. It is declared to be a pointer to a defined type FILE. Note that FILE is defined in **stdio.h** (see Figure 1-2).

This program fragment shows how **gets** is used.

```
#include <stdio.h>

main()
{
    char sarray[80];

    for(;;)
    {
        if (gets(sarray) != NULL)
        .
        .   /* Do something with the string */
        .
    }
}
```

The DESCRIPTION section of the **gets** manual page says, "**gets** reads characters from the standard input...." The standard input is usually the terminal you work from or a process that sends **gets** its output. The standard input is defined in **stdio.h** as shown in Figure 1-2.

```
#ifndef _NFILE
#define _NFILE          20

#define BUFSIZ          1024
#define _SBFSIZ 8

typedef struct {
            int         _cnt;
            unsigned char           *_ptr;
            unsigned char           *_base;
            char        _flag;
            char        _file;
} FILE;

#define _IOFBF          0000  /* _IOLBF means that a file's output  */
#define _IOREAD         0001  /* will be buffered line by line.     */
#define _IOWRT          0002  /* In addition to being flags, _IONBF,*/
#define _IONBF          0004  /* _IOLBF and IOFBF are possible      */
#define _IOMYBUF        0010  /* values for "type" in setvbuf.      */
#define _IOEOF          0020
#define _IOERR          0040
#define _IOLBF          0100
#define _IORW           0200

#ifndef NULL
#define NULL            0
#endif
#ifndef EOF
#define EOF (-1)
#endif
#define stdin                       (&_iob[0])
#define stdout                      (&_iob[1])
#define stderr                      (&_iob[2])

#define _bufend(p)      _bufendtab[(p)->_file]
#define _bufsiz(p)      (_bufend(p) - (p)->_base)
```

Figure 1-2. A version of stdio.h.

```
#ifndef lint
#define getc(p)  (--(p)->_cnt < 0 ? _filbuf(p) : (int) *(p)->_ptr++)
#define putc(x, p)    (--(p)->_cnt < 0 ?
        _flsbuf((unsigned char) (x), (p)) :
        (int) (*(p)->_ptr++ = (unsigned char) (x)))
#define getchar()     getc(stdin)
#define putchar(x)    putc((x), stdout)
#define clearerr(p)   ((void) ((p)->_flag &= ~(_IOERR | _IOEOF)))
#define feof(p)   ((p)->_flag & _IOEOF)
#define ferror(p)     ((p)->_flag & _IOERR)
#define fileno(p)     (p)->_file
#endif

extern FILE  _iob[_NFILE];
extern FILE  *fopen(), *fdopen(), *freopen(), *popen(), *tmpfile();
extern long  ftell();
extern void  rewind(), setbuf();
extern char  *ctermid(),*cuserid(),*fgets(),*gets(),*tempnam(),*tmpnam();
extern unsigned char *_bufendtab[];

#define L_ctermid     9
#define L_cuserid     9
#define P_tmpdir "/usr/tmp/"
#define L_tmpnam (sizeof(P_tmpdir) + 15)
#endif
```

Figure 1-2. A version of stdio.h (cont.).

1.4.4 Header Files and Libraries

In the earlier parts of this chapter there have been frequent references to **stdio.h**, and a version of the file itself is shown in Figure 1-2. **stdio.h** is the most commonly used header file in the operating system/C environment, but there are many others. Header files carry definitions and declarations that are used by more than one function. Header filenames traditionally have the suffix **.h** and are brought into a program at compile time by the C preprocessor. The preprocessor does this because it interprets the **#include** statement in your program as a directive, as indeed it is. All keywords preceded by a pound sign (#) at the beginning of the line are treated as preprocessor directives. The two most commonly used directives are **#include** and **#define**. We have already seen that the **#include** directive is used to call in (and process) the contents of the named file. The **#define** directive is used to replace a name with a token-string. For example, the following line sets to 20 the number of files a program can have open at one time.

```
#define _NFILE   20
```

See the **cpp**(1) manual page for the complete list of directives.

The *Reference Manual* lists about 45 different **.h** files. The format of the **#include** statement for all these shows the file name enclosed in angle brackets (<>), as in:

```
#include <stdio.h>
```

The angle brackets tell the C preprocessor to look in the standard places for the file. In most systems, the standard place is in the **/usr/include** directory. If you have some definitions or external declarations that you want to make available in several files, you can create a **.h** file with any editor, store it in a convenient directory, and make it the subject of a **#include** statement such as the following:

```
#include "../defs/rec.h"
```

It is necessary, in this case, to provide the relative pathname of the file and to enclose it in quotation marks (""). Fully-qualified pathnames (those that begin with /) can create portability and organizational problems. An alternative to long or fully-qualified pathnames is to use the **-I***dir* preprocessor option when you compile the program. This option directs the preprocessor to search for **#include** files whose names are enclosed in quotation marks, first in the directory of the file being compiled, then in the directories named in the **–I** option(s), and finally in directories on the standard list.

In addition, all **#include** files whose names are enclosed in angle brackets (< >) are first searched for in the list of directories named in the **–I** option and finally in the directories on the standard list.

1.4.5 Object File Libraries

It is common practice in DYNIX/ptx system computers to keep modules of compiled code (object files) in archives; by convention, these files are designated by a **.a** suffix. From the *Reference Manual*, system calls from Section 2 and the subroutines in Section 3 (3C and 3S), that are functions (as distinct from macros), are kept in an archive file named **libc.a**. In most systems, **libc.a** is found in the directory **/lib**. Many systems also have a directory **/usr/lib**. Where both **/lib** and **/usr/lib** occur, **/usr/lib** usually holds archives that are related to specific applications.

During the link edit phase of the compilation and link edit process, copies of some of the object modules in an archive file are loaded with your executable code. By default, the **cc** command that invokes the C compilation searched by default, you do it by naming them explicitly on the command line with the **–l** option. The format of the **–l** option is **–l***x* where *x* is the library name, and can be up to nine characters long. For example, if your program includes functions from the **curses** screen control package, the option

–lcurses

will cause the link editor to search for */lib/libcurses.a* or */usr/lib/libcurses.a* and use the first one it finds to resolve references in your program.

In cases where you want to direct the order in which archive libraries are searched, you may use the **–L** *dir* option. Assuming the **–L** option appears on the command line ahead of the **–l** option, it directs the link editor to search the named directory for *libx*.**a** before looking in */lib* and */usr/lib*. This is particularly useful if you are testing a new version of a function that already exists in an archive in a standard directory. Once a reference search is resolved, the link editor stops looking. That's why the **–L** option, if used, should appear on the command line ahead of any **–l** specification.

1.4.6 Input/Output

We discussed I/O earlier in this chapter in connection with system calls and subroutines. A whole set of subroutines constitutes the C language standard I/O package, and there are several system calls that deal with the same area. This section discusses the subject in more detail and describes how to deal with input and output concerns in C programs. I/O encompasses the following tasks:

- Creating and sometimes removing files
- Opening and closing files used by your program
- Transferring information from a file to your program (reading)
- Transferring information from your program to a file (writing)

This section describes some of the subroutines you might use to transfer information, but the emphasis will be on dealing with files.

Three Files You Always Have

Programs are permitted to have several files open simultaneously. The number may vary from system to system; the most common maximum is 20. _NFILE in **stdio.h** specifies the number of standard I/O FILEs a program is permitted to have open.

Any program automatically starts off with three files. If you will look again at Figure 1-2, about midway through you will see that **stdio.h** contains three **#define** directives that equate **stdin**, **stdout**, and **stderr** to the address of _iob[0], _iob[1], and _iob[2], respectively. The array _iob holds information dealing with the way standard I/O handles streams. It is a representation of the open file table in the control block for your program. The position in the array is held by a digit that is also known as the file descriptor. The default in DYNIX/ptx systems is to associate all three of these files with your terminal.

Functions and macros that deal with **stdin** or **stdout** can be used in your program with no further need to open or close files. For example, **gets**, shown in Figure 1-1, reads a string from **stdin**; **puts** writes a null-terminated string to **stdout**. There are others that do the same (in slightly different ways: one character at a time, formatted, and so on). You can specify that output be directed to **stderr** by using a function such as **fprintf**. **fprintf** works the same as **printf** except that it delivers its formatted output to a named stream, such as **stderr**. You can use the shell's redirection feature on the command line to read from or write into a named file. If you want to

separate error messages from ordinary output being sent to **stdout**, you can do it by using one function to handle the ordinary output and a variation of the same function that names the stream to handle error messages.

Named Files

Any files other than **stdin**, **stdout**, and **stderr** that are to be used by your program must be explicitly connected by you before the file can be read from or written to. This can be done using the standard library routine **fopen**. **fopen** takes a pathname (which is the name by which the file is known to the file system), asks the system to keep track of the connection, and returns a pointer that you then use in functions that do the reads and writes.

A structure is defined in **stdio.h** with a type of FILE. In your program you need to have a declaration such as the following:

```
FILE *fin;
```

The declaration says that **fin** is a pointer to a FILE. You can then assign the name of a particular file to the pointer with a statement in your program like this:

```
fin = fopen("filename", "r");
```

where **filename** is the pathname to open. The **"r"** means that the file is to be opened for reading. This argument is known as the **mode**. As you might suspect, there are modes for reading, writing, and both reading and writing. Actually, the file open function is often included in an **if** statement such as:

```
if ((fin = fopen("filename", "r")) == NULL)
  (void)fprintf(stderr,"%s: Unable to open input file %s\n",
  argv[0],"filename");
```

This statement takes advantage of the fact that **fopen** returns a NULL pointer if it can't open the file.

Once the file has been successfully opened, the pointer **fin** is used in functions (or macros) to refer to the file. For example, the following sequence brings in one character from the file into an integer variable called **c** each time **getc** is called:

```
int c;
c = getc(fin);
```

The variable **c** is declared as an integer even though we are reading characters because the function **getc()** returns an integer. Getting a character is often incorporated into some flow-of-control mechanism that reads through the

file until EOF is returned, such as:

```
while ((c = getc(fin)) != EOF)
      .
      .
      .
```

EOF, NULL, and the macro **getc** are all defined in **stdio.h**. **getc** and others that make up the standard I/O package keep advancing a pointer through the buffer associated with the file; the operating system and the standard I/O subroutines are responsible for seeing that the buffer is refilled (or written to the output file if you are producing output) when the pointer reaches the end of the buffer. All these mechanics are invisible to the program and the programmer.

The function **fclose** is used to break the connection between the pointer in your program and the pathname. The pointer may then be associated with another file by another call to **fopen**. This re-use of a file descriptor for a different stream may be necessary if your program has many files to open. For output files it is good to issue an **fclose** call because the call makes sure that all output has been sent from the output buffer before disconnecting the file. The system call **exit** closes all open files for you. It also gets you completely out of your process, however, so it is only safe to use when you are sure you are completely finished.

Low-Level I/O and Why You Shouldn't Use It

The term low-level I/O is used to refer to the process of using system calls from Section 2 of the *Reference Manual* rather than the functions and subroutines of the standard I/O package. We are going to postpone, until Chapter 3 of this guide, any discussion of when this might be advantageous. If you find as you go through the information in this chapter that it is a good fit with the objectives you have as a programmer, it is a safe assumption that you can work with C language programs for a good many years without ever having a real need to use system calls to handle your I/O and file accessing problems. The reason low-level I/O is perilous is because it is more system-dependent. Your programs are less portable and probably no more efficient.

1.4.7 System Calls for Environment or Status Information

Under some circumstances you might want to be able to monitor or control the environment in your computer. There are system calls that can be used for this purpose. Some of them are shown in Table 1-5 (see Section (2) in the *Reference Manual* for details).

Table 1-5
Environment and Status System Calls

Function Name(s)		Purpose
chdir		Change working directory
chmod		Change access permission of a file
chown		Change owner and group of a file
getpid	**getpgrp getppid**	Get process IDs
getuid	**geteuid getgid**	Get user IDs
ioctl		Control device
link	**unlink**	Add or remove a directory entry
mount	**umount**	Mount or unmount a file system
nice		Change priority of a process
stat	**fstat**	Get file status
time		Get time
ulimit		Get and set user limits
uname		Get name of current DYNIX/ptx system

As you can see, many of the functions shown in Table 1-5 have equivalent system shell commands. Shell commands can easily be incorporated into shell scripts to accomplish the monitoring and control tasks you may need to do. The functions are available, however, and may be used in C programs as part of the operating system/C Language interface. They are documented in Section 2 of the *Reference Manual.*

1.4.8 Processes

Whenever you execute a command in the system, you are initiating a process that is numbered and tracked by the operating system. A flexible feature of the operating system is that processes can be generated by other processes. This happens more than you might ever be aware of. For example, when you log in to your system you are running a process, very probably the shell. If you then use an editor such as **vi**, take the option of invoking the shell from **vi**, and execute the **ps** command, you will see a display something like that in Table 1-6 (which shows the results of a **ps –f** command):

Table 1-6
Process Status

UID	PID	PPID	C	STIME	TTY	TIME	COMMAND
abc	24210	1	0	06:13:14	tty29	0:05	**–sh**
abc	24631	24210	0	06:59:07	tty29	0:13	**vi c2.uli**
abc	28441	28358	80	09:17:22	tty29	0:01	**ps –f**
abc	28358	24631	2	09:15:14	tty29	0:01	**sh –i**

As you can see, user abc (who went through the steps described above) now has four processes active. It is an interesting exercise to trace the chain that is shown in the Process ID (PID) and Parent Process ID (PPID) columns. The shell that was started when user abc logged on is Process 24210; its parent is the initialization process (Process ID 1). Process 24210 is the parent of Process 24631, and so on.

The four processes in the table above are all system shell-level commands, but you can spawn new processes from your own program. (Actually, when you issue the command from your terminal to execute a program, you are asking the shell to start another process: the process being your executable object module with all the functions and subroutines that were made a part of it by the link editor.)

Overlooking the case where your program is itself an interactive application with diverse choices for the user, your program may need to run one or more other programs based on conditions it encounters in its own processing. (for example, if it's the end of the month, go do a trial balance). The usual reasons why it might not be practical to create one huge executable are as follows:

- The load module may get too big to fit in the maximum process size for your system; or it may fit, but still be so large that it degrades the performance of the entire system (other users are affected).
- You may not have control over the object code of all the other modules you want to include.

Suffice it to say, there are legitimate reasons to create new processes. There are three ways to do it:

- **system**(3S)—Request the shell to execute a command
- **exec**(2)—Stop this process and start another
- **fork**(2)—Start an additional copy of this process

system(3S)

The formal declaration of the **system** function looks like this:

```
#include <stdio.h>

int system(string)
char *string;
```

The function asks the shell to treat the string as a command line. The string can therefore be the name and arguments of any executable program or system shell command. If the exact arguments vary from one execution to the next, you may want to use **sprintf** to format the string before issuing the **system** command. When the command has finished running, **system** returns the shell exit status to your program. Execution of your program waits for the completion of the command initiated by **system** and then picks up again at the next executable statement.

exec(2)

exec is the name of a family of functions that includes **exec**, **execv**, **execle**, **execve**, **execlp**, and **execvp**. They all have the function of transforming the calling process into a new process. The reason for the variety is to provide different ways of pulling together and presenting the arguments of the function. An example of one version (**execl**) might be:

```
execl("/bin/prog2", "prog", progarg1, progarg2, (char *)0);
```

For **execl** the argument list is:

/bin/prog2	Path name of the new process file
prog	The name the new process gets in its argv[0]
progarg1, progarg2	Arguments to *prog2* as char *'s
(char *)0	A null char pointer to mark the end of the arguments

Check the manual page in the *Reference Manual* for the rest of the details. The key point of the **exec** family is that there is no return from a successful execution: the calling process is finished, the new process overlays the old. The new process also takes over the Process ID and other attributes of the old process. If the call to **exec** is unsuccessful, control is returned to your program with a return value of –1. You can check **errno** (see below) to learn why it failed.

fork(2)

The **fork** system call creates a new process that is an exact copy of the calling process. The new process is known as the child process; the caller is known as the parent process. The one major difference between the two processes is that the child gets its own unique process ID. When the **fork** process has completed successfully, it returns a 0 to the child process and the child's process ID to the parent. If the idea of having two identical processes seems a odd, consider this:

- Because the return value is different between the child process and the parent, the program can contain the logic to determine different paths.
- The child process could issue an **exec** for an entirely different program.
- The parent process, knowing this, would issue a **wait** until it gets word that that process is finished.

To accomplish this, your code might include statements like this:

```
#include <errno.h>

int ch_stat, ch_pid, status;
char *progarg1;
char *progarg2;
void exit();
extern int errno;
```

```
if ((ch_pid = fork()) < 0)
  {
 /* Could not fork...
    check errno
 */
  }
  else if (ch_pid == 0)      /* child */
  {
 (void)execl("/bin/prog2","prog",progarg1,progarg2,(char*)0);
 exit(2);/* execl() failed */
     }
  else              /* parent */
  {
 while ((status = wait(&ch_stat)) != ch_pid)
 {
     if (status < 0 && errno == ECHILD)
       break;
     errno = 0;
 }
  }
```

Because the child process ID is taken over by the new **exec**'d process, the parent knows the ID. This is a way of leaving one program to run another, returning to the point in the first program where processing left off. This is exactly what the **system**(3S) function does. As a matter of fact, **system** accomplishes it through this same procedure of **fork**ing and **exec**ing, with a **wait** in the parent.

Keep in mind that the fragment of code above includes a minimum amount of checking for error conditions. There is also potential confusion about open files and which program is writing to a file. Leaving out the possibility of named files, the new process created by the **fork** or **exec** has the three standard files that are automatically opened: **stdin**, **stdout**, and **stderr**. If the parent has buffered output that should appear before output from the child, the buffers must be flushed before the fork. Also, if the parent and the child process both read input from a stream, whatever is read by one process will be lost to the other. That is, once something has been delivered from the input buffer to a process, the pointer has moved on.

Pipes

The idea of using pipes, a connection between the output of one program and the input of another, when working with commands executed by the shell is well established in the system environment. For example, to learn the number of archive files in your system, you might enter a command like:

echo /lib/*.a /usr/lib/*.a | wc -w

This command first echoes all the files in **/lib** and **/usr/lib** that end in **.a**, then pipes the results to the **wc** command, which counts their number.

A feature of the operating system/C Language interface is the ability to establish pipe connections between your process and a command to be executed by the shell, or between two cooperating processes. The first uses the **popen**(3S) subroutine that is part of the standard I/O package; the second requires the system call **pipe**(2).

popen is similar in concept to the **system** subroutine in that it causes the shell to execute a command. The difference is that once having invoked **popen** from your program, you have established an open line to a concurrently running process through a stream. You can send characters or strings to this stream with standard I/O subroutines just as you would to **stdout** or to a named file. The connection remains open until your program invokes the companion **pclose** subroutine. A common application of this technique might be a pipe to a printer spooler. For example:

```
#include <stdio.h>

main()
{
    FILE *pptr;
    char *outstring;

    if ((pptr = popen("lp","w")) != NULL)
    {
        for(;;)
        {   .
            .   /* Organize output */
            .
            (void)fprintf(pptr, "%s\n", outstring);
            .
            .
            .
        }
```

```
        .
        .
        .
        pclose(pptr);
            }
            .
            .
            .
}
```

1.4.9 Error Handling

Within your C programs, you must determine the appropriate level of checking for valid data and for acceptable return codes from functions and subroutines. If you use any of the system calls described in Section 2 of the *Reference Manual*, you have a way to find the probable cause of a bad return value.

System calls that are not able to complete successfully almost always return a value of -1 to your program. (If you look through the system calls in Section 2, you will see that there are a few calls for which no return value is defined, but they are the exceptions.) In addition to the -1 that is returned to the program, the unsuccessful system call places an integer in an externally declared variable, **errno**. You can determine the value in **errno** if your program contains the following statement:

```
#include <errno.h>
```

The value in **errno** is not cleared on successful calls, so your program should check it only if the system call returned a -1. The errors are described in **intro**(2) in the *Reference Manual.*

The subroutine **perror**(3C) can be used to print an error message (on **stderr**) based on the value of **errno**.

1.4.10 Signals and Interrupts

Signals and *interrupts* are two words for the same thing. Both words refer to messages passed by the operating system to running processes. Generally, the effect is to cause the process to stop running. Some signals are generated if the process attempts to do something illegal; others can be initiated by a user against his or her own processes, or by the superuser against any process.

There is a system call, **kill**, that you can include in your program to send signals to other processes running under your user-id. The format for the **kill** call is:

```
kill(pid, sig)
```

where **pid** is the process number against which the call is directed, and **sig** is an integer from 1 to 19 that shows the intent of the message. The name "kill" is something of an overstatement; not all the messages have a "drop dead" meaning. Some of the available signals are shown in the following example as they are defined in **<sys/signal.h>**.

```
#define  SIGHUP     1    /* hangup */
#define  SIGINT     2    /* interrupt (rubout) */
#define  SIGQUIT    3    /* quit (ASCII FS) */
#define  SIGILL     4    /* illegal instruction (not reset when caught)*/
#define  SIGTRAP    5    /* trace trap (not reset when caught) */
#define  SIGIOT     6    /* IOT instruction */
#define  SIGABRT    6    /* used by abort, replace SIGIOT in the future */
#define  SIGEMT     7    /* EMT instruction */
#define  SIGFPE     8    /* floating point exception */
#define  SIGKILL    9    /* kill (cannot be caught or ignored) */
#define  SIGBUS     10   /* bus error */
#define  SIGSEGV    11   /* segmentation violation */
#define  SIGSYS     12   /* bad argument to system call */
#define  SIGPIPE    13   /* write on a pipe with no one to read it */
#define  SIGALRM    14   /* alarm clock */
#define  SIGTERM    15   /* software termination signal from kill */
#define  SIGUSR1    16   /* user defined signal 1 */
#define  SIGUSR2    17   /* user defined signal 2 */
#define  SIGCLD     18   /* death of a child */
#define  SIGPWR     19   /* power-fail restart */

#define  SIGPOLL 20 /* pollable event occurred */

#define  NSIG       21   /* The valid signal number is from 1 to NSIG-1 */
#define  MAXSIG     32   /* size of u_signal[], NSIG-1 <= MAXSIG*/
                         /* MAXSIG is larger than we need now. */
                         /* In the future, we can add more signal */
                         /* number without changing user.h */
```

The **signal**(2) system call is designed to let you code methods of dealing with incoming signals. You have three choices. You can accept whatever the default action is for the signal, have your program ignore the signal, or write a function of your own to deal with it.

1.5 Analysis/Debugging

The system provides several commands designed to help you discover the causes of problems in programs and to learn about potential problems.

1.5.1 Sample Program

To illustrate how these commands are used and the type of output they produce, we have constructed a sample program that opens and reads an input file and performs one to three subroutines according to options specified on the command line. This program does not do anything you couldn't do quite easily on your pocket calculator, but it does serve to illustrate some points. The source code is shown in the next example. The header file, **recdef.h**, is shown at the end of the source code.

The output produced by the various analysis and debugging tools illustrated in this section may vary slightly from one installation to another. The *Reference Manual* is a good source of additional information about the contents of the reports.

```
            /* Main module -- restate.c */

#include <stdio.h>
#include "recdef.h"

#define TRUE  1
#define FALSE 0

main(argc, argv)
int argc;
char *argv[];
{
    FILE *fopen(), *fin;
    void exit();
    int getopt();
    int oflag = FALSE;
    int pflag = FALSE;
    int rflag = FALSE;
    int ch;
    struct rec first;
    extern int opterr;
    extern float oppty(), pft(), rfe();

if (argc < 2)
{
```

```
    (void) fprintf(stderr, "%s: Must specify option\n",argv[0]);
    (void) fprintf(stderr, "Usage: %s -rpo\n", argv[0]);
    exit(2);
}

opterr = FALSE;
while ((ch = getopt(argc,argv,"opr")) != EOF)
{
    switch(ch)
    {
    case 'o':
        oflag = TRUE;
        break;
    case 'p':
        pflag = TRUE;
        break;
    case 'r':
        rflag = TRUE;
        break;
    default:
        (void) fprintf(stderr, "Usage: %s -rpo\n",argv[0]);
        exit(2);
    }
}
if ((fin = fopen("info","r")) == NULL)
{
(void) fprintf(stderr, "%s: cannot open input file %s\n",argv[0],"info");
exit(2);
}

if (fscanf(fin, "%s%f%f%f%f%f%f", first.pname,&first.ppx,
&first.dp,&first.i,&first.c,&first.t,&first.spx) != 7)
{
  (void) fprintf(stderr,"%s: cannot read first record from %s\n",
    argv[0],"info");
  exit(2);
}

printf("Property: %s\n",first.pname);

if(oflag)
    printf("Opportunity Cost: $%#5.2f\n",oppty(&first));
```

```
if(pflag)
    printf("Anticipated Profit(loss): $%#7.2f\n",pft(&first));

if(rflag)
    printf("Return on Funds Employed: %#3.2f%%\n",rfe(&first));
}
        /* End of Main Module -- restate.c */

        /* Opportunity Cost -- oppty.c */
#include "recdef.h"

float
oppty(ps)
struct rec *ps;
{
    return(ps->i/12 * ps->t * ps->dp);
}

        /* Profit -- pft.c */

#include "recdef.h"

float
pft(ps)
struct rec *ps;
{
    return(ps->spx - ps->ppx + ps->c);
}

        /* Return on Funds Employed -- rfe.c */

#include "recdef.h"

float
rfe(ps)
struct rec *ps;
{
    return(100 * (ps->spx - ps->c) / ps->spx);
}

            /* Header File -- recdef.h */

struct rec {      /* To hold input  */
    char pname[25];
    float ppx;
    float dp;
    float i;
```

```
    float c;
    float t;
    float spx;
} ;
```

1.5.2 cflow

cflow produces a chart of the external references in C, **yacc**, **lex**, and assembly language files. Using the modules of our sample program, the command

cflow restate.c oppty.c pft.c rfe.c

produces the output shown in the next example.

```
1   main: int(), <restate.c 11>
2       fprintf: <>
3       exit: <>
4       getopt: <>
5       fopen: <>
6       fscanf: <>
7       printf: <>
8       oppty: float(), <oppty.c 7>
9       pft: float(), <pft.c 7>
10      rfe: float(), <rfe.c 8>
```

The **–r** option looks at the caller:callee relationship from the other side. It produces the output shown here:

```
1   exit: <>
2       main : <>
3   fopen: <>
4       main : 2
5   fprintf: <>
6       main : 2
7   fscanf: <>
8       main : 2
9   getopt: <>
10      main : 2
11  main: int(), <restate.c 11>
12  oppty: float(), <oppty.c 7>
13      main : 2
14  pft: float(), <pft.c 7>
15      main : 2
16  printf: <>
```

```
17      main : 2
18  rfe: float(), <rfe.c 8>
19      main : 2
```

The **-ix** option causes external and static data symbols to be included. Our sample program has only one such symbol, **opterr**. The output is shown here:

```
1   main: int(), <restate.c 11>
2       fprintf: <>
3       exit: <>
4       opterr: <>
5       getopt: <>
6       fopen: <>
7       fscanf: <>
8       printf: <>
9       oppty: float(), <oppty.c 7>
10      pft: float(), <pft.c 7>
11      rfe: float(), <rfe.c 8>
```

Combining the **-r** and the **-ix** options produces the output shown here:

```
1   exit: <>
2       main : <>
3   fopen: <>
4       main : 2
5   fprintf: <>
6       main : 2
7   fscanf: <>
8       main : 2
9   getopt: <>
10      main : 2
11  main: int(), <restate.c 11>
12  oppty: float(), <oppty.c 7>
13      main : 2
14  opterr: <>
15      main : 2
16  pft: float(), <pft.c 7>
17      main : 2
18  printf: <>
19      main : 2
20  rfe: float(), <rfe.c 8>
21      main : 2
```

1.5.3 ctrace

ctrace lets you follow the execution of a C program statement by statement. **ctrace** takes a **.c** file as input and inserts statements in the source code to print out variables as each program statement is executed. You must direct the output of this process to a temporary **.c** file. The temporary file is then used as input to **cc**. When the resulting **a.out** file is executed, it produces output that can tell you a lot about what is going on in your program.

Options give you the ability to limit the number of times through loops. You can also include functions in your source file that turn the trace off and on so that you can limit the output to portions of the program that are of particular interest.

ctrace accepts only one source code file as input. To use our sample program to illustrate, it is necessary to execute the following four commands:

```
ctrace restate.c > ct.main.c
ctrace oppty.c > ct.op.c
ctrace pft.c > ct.p.c
ctrace rfe.c > ct.r.c
```

The names of the output files are completely arbitrary. Use any names that are convenient for you. The names must end in **.c**, since the files are used as input to the C compilation system.

```
cc –o ct.run ct.main.c ct.op.c ct.p.c ct.r.c
```

The following command produces the output shown in the next example:

```
ct.run –opr
```

This command causes the output to be directed to your terminal (**stdout**). It is probably a good idea to direct it to a file or to a printer so you can refer to it.

```
 8 main(argc, argv)
23      if (argc < 2)
        /* argc == 2 */
30      opterr = FALSE;
        /* FALSE == 0 */
        /* opterr == 0 */
31      while ((ch = getopt(argc,argv,"opr")) != EOF)
        /* argc == 2 */
```

```
        /* argv == 15729316 */
        /* ch == 111 or 'o' or "t" */
32      {
33          switch(ch)
            /* ch == 111 or 'o' or "t" */
35          case 'o':
36              oflag = TRUE;
                /* TRUE == 1 or "h" */
                /* oflag == 1 or "h" */
37              break;
48      }
31      while ((ch = getopt(argc,argv,"opr")) != EOF)
        /* argc == 2 */
        /* argv == 15729316 */
        /* ch == 112 or 'p' */
32      {
33          switch(ch)
            /* ch == 112 or 'p' */
38          case 'p':
39              pflag = TRUE;
                /* TRUE == 1 or "h" */
                /* pflag == 1 or "h" */
40              break;
48      }
31      while ((ch = getopt(argc,argv,"opr")) != EOF)
        /* argc == 2 */
        /* argv == 15729316 */
        /* ch == 114 or 'r' */
32      {
33          switch(ch)
            /* ch == 114 or 'r' */
41          case 'r':
42              rflag = TRUE;
                /* TRUE == 1 or "h" */
                /* rflag == 1 or "h" */
43              break;
48      }
31      while ((ch = getopt(argc,argv,"opr")) != EOF)
        /* argc == 2 */
        /* argv == 15729316 */
        /* ch == -1 */
49      if ((fin = fopen("info","r")) == NULL)
```

```
      /* fin == 140200 */
54    if (fscanf(fin, "%s%f%f%f%f%f%f",first.pname,&first.ppx,
  &first.dp,&first.i,&first.c,&first.t,&first.spx)!= 7)
      /* fin == 140200 */
      /* first.pname == 15729528 */
61    printf("Property: %s0,first.pname);
      /* first.pname == 15729528 or "Linden_Place" */
  Property: Linden_Place

63    if(oflag)
      /* oflag == 1 or "h" */
64       printf("Opportunity Cost: $%#5.2f0,oppty(&first));
 5 oppty(ps)
 8    return(ps->i/12 * ps->t * ps->dp);
      /* ps->i == 1069044203 */
      /* ps->t == 1076494336 */
      /* ps->dp == 1088765312 */     Opportunity Cost: $4476.87

66    if(pflag)
      /* pflag == 1 or "h" */
67       printf("Anticipated Profit(loss): $%#7.2f0,pft(&first));
 5 pft(ps)
 8    return(ps->spx - ps->ppx + ps->c);
      /* ps->spx == 1091649040 */
      /* ps->ppx == 1091178464 */
      /* ps->c == 1087409536 */ Anticipated Profit(loss): $85950.00

 9    return(100 * (ps->spx - ps->c) / ps->spx);
      /* ps->spx == 1091649040 */
      /* ps->c == 1087409536 */ Return on Funds Employed: 94.00%

  /* return */
```

Using a program that runs successfully is not the best way to demonstrate **ctrace**. It would be more helpful to have an error in the operation that could be detected by **ctrace**. This utility might be most useful in cases where the program runs to completion, but produces unexpected output.

1.5.4 cxref

cxref analyzes a group of C source code files and builds a cross-reference table of the automatic, static, and global symbols in each file.

The command

cxref –c –o cx.op restate.c oppty.c pft.c rfe.c

produces the output shown in in the next example in a file named, in this case, **cx.op**. The **–c** option causes the reports for the four **.c** files to be combined in one cross-reference file.

```
restate.c:

oppty.c:

pft.c:

rfe.c:

 SYMBOL        FILE              FUNCTION LINE

BUFSIZ      /usr/include/stdio.h        --*9
EOF         /usr/include/stdio.h        --49 *50
            restate.c           --      31
FALSE       restate.c           --      *6  15  16  17  30
FILE        /usr/include/stdio.h        --*29  73  74
            restate.c           main    12
L_ctermid   /usr/include/stdio.h        --*80
L_cuserid   /usr/include/stdio.h        --*81
L_tmpnam    /usr/include/stdio.h        --*83
NULL        /usr/include/stdio.h        --46 *47
            restate.c           --      49
P_tmpdir    /usr/include/stdio.h        --*82
TRUE        restate.c           --      *5  36  39  42
_IOEOF      /usr/include/stdio.h        --*41
_IOERR      /usr/include/stdio.h        --*42
_IOFBF      /usr/include/stdio.h        --*36
_IOLBF      /usr/include/stdio.h        --*43
_IOMYBUF    /usr/include/stdio.h        --*40
_IONBF      /usr/include/stdio.h        --*39
_IOREAD     /usr/include/stdio.h        --*37
_IORW       /usr/include/stdio.h        --*44
_IOWRT      /usr/include/stdio.h        --*38
```

```
_NFILE      /usr/include/stdio.h        --2 *3  73
_SBFSIZ     /usr/include/stdio.h        --*16
_base       /usr/include/stdio.h        --*26
_bufend()
            /usr/include/stdio.h        --*57
_bufendtab  /usr/include/stdio.h        --*78
_bufsiz()
            /usr/include/stdio.h        --*58
_cnt        /usr/include/stdio.h        --*20
_file       /usr/include/stdio.h        --*28
_flag       /usr/include/stdio.h        --*27
_iob        /usr/include/stdio.h        --*73
            restate.c           main    25 26 45 51 57
_ptr        /usr/include/stdio.h        --*21
argc        restate.c           --      8
            restate.c           main    *9  23  31
argv        restate.c           --      8
            restate.c           main    *10 25 26 31 45 51 57
c           ./recdef.h          --      *6
            pft.c               pft     8
            restate.c           main    55
            rfe.c               rfe     9
ch          restate.c           main    *18  31  33
clearerr()
            /usr/include/stdio.h        --*67
ctermid()
            /usr/include/stdio.h        --*77
cuserid()
            /usr/include/stdio.h        --*77
dp          ./recdef.h                  --*4
            oppty.c             oppty   8
            restate.c           main    55
exit()
            restate.c           main    *13  27  46  52  58
fdopen()
            /usr/include/stdio.h        --*74
feof()
            /usr/include/stdio.h        --*68
ferror()
            /usr/include/stdio.h        --*69
fgets()
            /usr/include/stdio.h        --*77
```

```
fileno()
            /usr/include/stdio.h        --*70
fin         restate.c           main    *12  49  54
first       restate.c           main    *19 54 55 61 64 67 70
fopen()
            /usr/include/stdio.h        --*74
            restate.c           main    12  49
fprintf     restate.c           main    25 26 45 51 57
freopen()
            /usr/include/stdio.h        --*74
fscanf      restate.c           main    54
ftell()
            /usr/include/stdio.h        --*75
getc()
            /usr/include/stdio.h        --*61
getchar()
            /usr/include/stdio.h        --*65
getopt()
            restate.c           main    *14  31
gets()
            /usr/include/stdio.h        --*77
i           ./recdef.h          --      *5
            oppty.c             oppty   8
            restate.c           main    55
lint        /usr/include/stdio.h        --60
main()
            restate.c           --      *8
oflag       restate.c           main    *15  36  63
oppty()
            oppty.c             --      *5
            restate.c           main    *21  64
opterr      restate.c           main    *20  30
p           /usr/include/stdio.h        --*57 *58 *61 62
*62  63  64  67 *67  68 *68  69 *69  70 *70
pdp11       /usr/include/stdio.h        --11
pflag       restate.c           main    *16  39  66
pft()
            pft.c               --      *5
            restate.c           main    *21  67
pname       ./recdef.h          --      *2
            restate.c           main    54  61
popen()
```

```
            /usr/include/stdio.h        --*74
ppx         ./recdef.h          --      *3
            pft.c               pft     8
            restate.c           main    54
printf      restate.c           main    61  64  67  70
ps          oppty.c             --      5
            oppty.c             oppty   *6  8
            pft.c               --      5
            pft.c               pft     *6  8
            rfe.c               --      6
            rfe.c               rfe     *7  9
putc()
            /usr/include/stdio.h        --*62
putchar()
            /usr/include/stdio.h        --*66
rec         ./recdef.h          --      *1
            oppty.c             oppty   6
            pft.c               pft     6
            restate.c           main    19
            rfe.c               rfe     7
rewind()
            /usr/include/stdio.h        --*76
rfe()
            restate.c           main    *21  70
            rfe.c               --      *6
rflag       restate.c           main    *17  42  69
setbuf()
            /usr/include/stdio.h        --*76
spx         ./recdef.h          --      *8
            pft.c               pft     8
            restate.c           main    55
            rfe.c               rfe     9
stderr      /usr/include/stdio.h        --*55
            restate.c           --      25  26  45  51  57
stdin       /usr/include/stdio.h        --*53
stdout      /usr/include/stdio.h        --*54
t           ./recdef.h          --      *7
            oppty.c             oppty   8
            restate.c           main    55
tempnam()
            /usr/include/stdio.h        --*77
tmpfile()
```

```
                /usr/include/stdio.h        --*74
tmpnam()
                /usr/include/stdio.h        --*77
u370            /usr/include/stdio.h        --5
u3b             /usr/include/stdio.h        --8  19
u3b5            /usr/include/stdio.h        --8  19
vax             /usr/include/stdio.h        --8  19
x               /usr/include/stdio.h        --*62  63  64  66 *66
```

1.5.5 lint

lint looks for features in a C program that are apt to cause execution errors, that are wasteful of resources, or that create problems of portability.

The command

lint restate.c oppty.c pft.c rfe.c

produces the output shown in the next example.

```
restate.c:
restate.c
==============
(71)  warning: main() returns random value to invocation environment
oppty.c:
pft.c:
rfe.c:
==============
function returns value which is always ignored
    printf
```

lint has options that will produce additional information. Check the **lint**(1) man page. The error messages give you the line numbers of some items you may want to review.

1.5.6 prof

prof produces a report on the amount of execution time spent in various portions of your program and the number of times each function is called. The program must be compiled with the **–p** option. When a program that was compiled with that option is run, a file called **mon.out** is produced. **mon.out** and **a.out** (or whatever name identifies your executable file) are input to the **prof** command.

The sequence of steps needed to produce a profile report for our sample program is as follows:

Step 1: Compile the programs with the **–p** option:

cc –p restate.c oppty.c pft.c rfe.c

Step 2: Run the program to produce a file **mon.out**.

a.out –opr

Step 3: Execute the **prof** command:

prof a.out

The example of the output of this last step is shown in Table 1-7. The figures may vary from one run to another. You will also notice that programs of very small size, like that used in the example, produce statistics that are not overly helpful.

Table 1-7
prof Output

%Time	Seconds	Cumsecs	#Calls	msec/call	Name
50.0	0.03	0.03	3	8.	fcvt
20.0	0.01	0.04	6	2.	atof
20.0	0.01	0.05	5	2.	write
10.0	0.00	0.05	1	5.	fwrite
0.0	0.00	0.05	1	0.	monitor
0.0	0.00	0.05	1	0.	creat
0.0	0.00	0.05	4	0.	printf
0.0	0.00	0.05	2	0.	profil
0.0	0.00	0.05	1	0.	fscanf
0.0	0.00	0.05	1	0.	_doscan
0.0	0.00	0.05	1	0.	oppty
0.0	0.00	0.05	1	0.	_filbuf
0.0	0.00	0.05	3	0.	strchr
0.0	0.00	0.05	1	0.	strcmp
0.0	0.00	0.05	1	0.	ldexp
0.0	0.00	0.05	1	0.	getenv
0.0	0.00	0.05	1	0.	fopen
0.0	0.00	0.05	1	0.	_findiop
0.0	0.00	0.05	1	0.	open
0.0	0.00	0.05	1	0.	main
0.0	0.00	0.05	1	0.	read
0.0	0.00	0.05	1	0.	strcpy
0.0	0.00	0.05	14	0.	ungetc

Table 1-7
prof Output (cont.)

%Time	Seconds	Cumsecs	#Calls	msec/call	Name
0.0	0.00	0.05	4	0.	_doprnt
0.0	0.00	0.05	1	0.	pft
0.0	0.00	0.05	1	0.	rfe
0.0	0.00	0.05	4	0.	_xflsbuf
0.0	0.00	0.05	1	0.	_wrtchk
0.0	0.00	0.05	2	0.	_findbuf
0.0	0.00	0.05	2	0.	isatty
0.0	0.00	0.05	2	0.	ioctl
0.0	0.00	0.05	1	0.	malloc
0.0	0.00	0.05	1	0.	memchr
0.0	0.00	0.05	1	0.	memcpy
0.0	0.00	0.05	2	0.	sbrk
0.0	0.00	0.05	4	0.	getopt

1.5.7 size

size produces information on the number of bytes occupied by the three sections (text, data, and bss) of a common object file when the program is brought into main memory to be run. Here are the results of one invocation of the **size** command with our object file as an argument.

```
11832 + 3872 + 2240 = 17944
```

Don't confuse this number with the number of characters in the object file that appears when you do an **ls –l** command. That figure includes the symbol table and other header information that is not used at run time.

1.5.8 strip

strip removes the symbol and line number information from a common object file. When you issue this command, the number of characters shown by the **ls –l** command approaches the figure shown by the **size** command, but still includes some header information that is not counted as part of the .text, .data, or .bss section. After the **strip** command has been executed, it is no longer possible to use the file with the **pdbx** command.

1.5.9 Pdbx

Pdbx is an interactive, symbolic debugger for use in debugging applications written in C, FORTRAN, Pascal, assembly language, or any combination of these languages. You can use Pdbx to debug both conventional, one-process applications and parallel applications (those consisting of two or more cooperating processes executing concurrently). You can also use Pdbx to debug applications that consist of multiple processes running different programs such as client/server applications.

1.6 Program Organizing Utilities

The following three utilities are helpful in keeping your programming work organized effectively.

1.6.1 The make Command

When you have a program that is made up of more than one module of code you begin to run into problems of keeping track of which modules are up-to-date and which need to be recompiled when changes are made in another module. The **make** command is used to ensure that dependencies between modules are recorded so that changes in one module result in the re-compilation of dependent programs. Even control of a program as simple as the one shown in the last example is made easier through the use of **make**.

The **make** utility requires a description file that you create with an editor. The description file (also referred to by its default name: **makefile**) contains the information used by **make** to keep a target file current. The target file is typically an executable program. A description file contains three types of information:

Dependency information	Tells the **make** utility the relationship between the modules that compose the target program.
Executable commands	Needed to generate the target program. **make** uses the dependency information to determine which executable commands should be passed to the shell for execution.
Macro definitions	Provide a shorthand notation within the description file to make maintenance easier. Macro definitions can be overridden by information from the command line when the **make** command is entered.

The **make** command works by checking the "last changed" time of the modules named in the description file. When **make** finds a component that has been changed more recently than modules that depend on it, the specified commands (usually compilations) are passed to the shell for execution.

The **make** command takes three kinds of arguments: options, macro definitions, and target filenames. If no description filename is given as an option on the command line, **make** searches the current directory for a file named **makefile** or **Makefile**. The next example shows a **makefile** for our sample program.

```
OBJECTS = restate.o oppty.o pft.o rfe.o
all: restate
restate: $(OBJECTS)
    $(CC) $(CFLAGS) $(LDFLAGS) $(OBJECTS) -o restate

$(OBJECTS): ./recdef.h

clean:
    rm -f $(OBJECTS)

clobber:   clean
    rm -f restate
```

The following things are worth noticing in this description file:

- It identifies the target, **restate**, as being dependent on the four object modules. Each of the object modules in turn is defined as being dependent on the header file, **recdef.h**, and by default, on its corresponding source file.
- A macro, OBJECTS, is defined as a convenient shorthand for referring to all of the component modules.

Whenever testing or debugging results in a change to one of the components of **restate**, for example, a command such as the following should be entered:

make CFLAGS=-g restate

This has been a very brief overview of the **make** utility. There is more on **make** in Chapter 3; a detailed description of **make** can be found in the *Programming Tools Guide.*

1.6.2 The Archive

The most common use of an archive file, although not the only one, is to hold object modules that make up a library. The library can be named on the link editor command line (or with a link editor option on the **cc** command line). This causes the link editor to search the symbol table of the archive file when attempting to resolve references.

The **ar** command is used to create an archive file, to manipulate its contents, and to maintain its symbol table. The structure of the **ar** command is a little different from the normal system arrangement of command line options. When you enter the **ar** command, you include a one-character key from the set **drqtpmx** that defines the type of action you intend. The key may be combined with one or more additional characters from the set **vuaibcls** that modify the way the requested operation is performed. The makeup of the command line is

ar -key [*posname*] *afile* [*name*]...

where *posname* is the name of a member of the archive and may be used with some optional key characters to make sure that the files in your archive are in a particular order. The *afile* argument is the name of your archive file. By convention, the suffix **.a** is used to indicate that the named file is an archive file. (**libc.a**, for example, is the archive file that contains many of the object files of the standard C subroutines.) One or more *names* may be furnished. These identify files that are subjected to the action specified in the *key*.

We can make an archive file to contain the modules used in our sample program, **restate**. The command to do this is as follows:

ar –rv rste.a restate.o oppty.o pft.o rfe.o

If these are the only **.o** files in the current directory, you can use shell metacharacters as follows: **ar –rv rste.a *.o**. Either command will produce this feedback:

```
a - restate.o
a - oppty.o
a - pft.o
a - rfe.o
ar: creating rste.a
```

The **nm** command is used to get a variety of information from the symbol table of common object files. The object files can be, but don't have to be, in an archive file. Table 1-8 shows the output of this command when executed with the **–f** (for full) option on the archive we just created. The object files were compiled with the **–g** option.

Table 1-8
nm Output, with –f Option

Symbols from rste.a[restate.o]						
Name	**Value**	**Class**	**Type**	**Size**	**Line**	**Section**
.0fake			strtag	struct	16	
restate.c		file				
_cnt	0	strmem	int			
_ptr	4	strmem	*Uchar			
_base	8	strmem	*Uchar			
_flag	12	strmem	char			
_file	13	strmem	char			
.eos		endstr		16		
rec		strtag	struct	52		
pname	0	strmem	char[25]	25		
ppx	28	strmem	float			
dp	32	strmem	float			
i	36	strmem	float			
c	40	strmem	float			
t	44	strmem	float			
spx	48	strmem	float			
.eos		endstr		52		
main	0	extern	int( )	520		.text
.bf	10	fcn			11	.text
argc	0	argm't	int			
argv	4	argm't	**char			
fin	0	auto	*struct-.0fake	16		
oflag	4	auto	int			
pflag	8	auto	int			
rflag	12	auto	int			
ch	16	auto	int			
first	20	auto	struct-rec	52		
.ef	518	fcn			61	.text
FILE		typdef	struct-.0fake	16		
.text	0	static		31	39	.text
.data	520	static			4	.data
.bss	824	static				.bss

Table 1-8
nm Output, with -f Option (cont.)

Symbols from rste.a[restate.o]						
Name	**Value**	**Class**	**Type**	**Size**	**Line**	**Section**
_iob	0	extern				
fprintf	0	extern				
exit	0	extern				
opterr	0	extern				
getopt	0	extern				
fopen	0	extern				
fscanf	0	extern				
printf	0	extern				
oppty	0	extern				
pft	0	extern				
rfe	0	extern				
oppty.c		file				
rec		strtag	struct	52		
pname	0	strmem	char[25]	25		
ppx	28	strmem	float			
dp	32	strmem	float			
i	36	strmem	float			
c	40	strmem	float			
t	44	strmem	float			
spx	48	strmem	float			
.eos		endstr		52		
oppty	0	extern	float()	64		.text
.bf	10	fcn			7	.text
ps	0	argm't	*struct-rec	52		
.ef	62	fcn			3	.text
.text	0	static		4	1	.text
.data	64	static				.data
.bss	72	static				.bss
pft.c		file				
rec		strtag	struct	52		

Table 1-8
nm Output, with –f Option (cont.)

Symbols from rste.a[pft.o]						
Name	**Value**	**Class**	**Type**	**Size**	**Line**	**Section**
pname	0	strmem	char[25]	25		
ppx	28	strmem	float			
dp	32	strmem	float			
i	36	strmem	float			
c	40	strmem	float			
t	44	strmem	float			
spx	48	strmem	float			
..eos		endstr		52		
pft	0	extern	float()	60		.text
..bf	10	fcn			7	.text
ps	0	argm't	*struct-rec	52		
..ef	58	fcn			3	.text
..text	0	static		4		.text
..data	60	static				.data
..bss	60	static				.bss
rfe.c		file				
rec		strtag	struct	52		
pname	0	strmem	char[25]	25		
ppx	28	strmem	float			
dp	32	strmem	float			
i	36	strmem	float			
c	40	strmem	float			
t	44	strmem	float			
spx	48	strmem	float			
.eos		endstr		52		
rfe	0	extern	float()	68		.text
.bf	10	fcn			8	.text
ps	0	argm't	*struct-rec	52		
.ef	64	fcn			3	.text
.text	0	static		4	1	.text
.data	68	static				.data
.bss	76	static				.bss

For **nm** to work on an archive file, all of the contents of the archive have to be object modules. If you have stored other things in the archive, you will get the following message when you try to execute the command:

```
nm: rste.a  bad magic
```

1.6.3 Use of SCCS by Single-User Programmers

The system Source Code Control System (SCCS) is a set of programs designed to keep track of different versions of programs. When a program has been placed under control of SCCS, only a single copy of any one version of the code can be retrieved for editing at a given time. When program code is changed and the program returned to SCCS, only the changes are recorded. Each version of the code is identified by its SID, or SCCS Identifying number. By specifying the SID when the code is extracted from the SCCS file, it is possible to return to an earlier version. If an early version is extracted with the intent of editing it and returning it to SCCS, a new branch of the development tree is started. The set of programs that make up SCCS appear as system commands. The commands are as follows:

admin
get
delta
prs
rmdel
cdc
what
sccsdiff
comb
val

It is most common to think of SCCS as a tool for project control of large programming projects. It is, however, entirely possible for any individual user of the operating system to set up a private SCCS system. An SCCS user's guide is available in the *Programming Tools Guide.*

Chapter 2
Application Programming Notes

2.1 Introduction 2-1

2.2 Application Programming 2-2
Numbers 2-2
Portability 2-2
Documentation 2-3

2.3 Language Selection 2-3
Influences 2-4
Special-Purpose Languages 2-4
What awk Is Like 2-4
How awk Is Used 2-5
Other Information About awk 2-5
What lex and yacc Are Like 2-6
How **lex** Is Used 2-7
Other Information About lex 2-8
How yacc Is Used 2-8
Other Information About yacc 2-10

2.4 Advanced Programming Tools 2-10
Memory Management 2-10
File and Record Locking 2-11
How File and Record Locking Works 2-12
lockf 2-13
Other Information 2-14
Interprocess Communications 2-14
IPC **get** Calls 2-15
IPC **ctl** Calls 2-15
IPC **op** Calls 2-15
Other Information 2-16
Programming Terminal Screens 2-16
curses 2-16
Other Information About curses and terminfo 2-17

2.5 Programming Support Tools 2-17
Link Edit Command Language 2-18
Other Information 2-18

Common Object File Format 2-18
Other Information 2-19
Libraries 2-19
The Object File Library 2-19
Common Object File Interface Macros (**ldfcn.h**) 2-21
The Math Library 2-24
Trigonometric Functions 2-25
Shared Libraries 2-28
Other Information About Shared Libraries 2-29
Pdbx — A Symbolic Debugger 2-29
Other Information About Pdbx 2-29
lint as a Portability Tool 2-29
Other Information 2-30

2.6 Project Control Tools 2-30
make 2-30
Other Information About make 2-31
SCCS 2-31
Other Information 2-32

2.7 Example — liber, A Library System 2-33

Tables

2-1 Object File Interface Macros 2-21
2-2 Trigonometric Functions 2-25
2-3 Hyperbolic Functions 2-26
2-4 Miscellaneous Functions 2-27

Chapter 2
Application Programming Notes

2.1 Introduction

This chapter deals with programming where the objective is to produce sets of programs (applications) that will run on your system.

The chapter begins with a discussion of how the ground rules change as you move up the scale from writing programs that are essentially Li your own private use (we have called this single-user programming) to working as a member of a programming team developing an application that is to be turned over to others to use.

There is a section on how the criteria for selecting appropriate programming languages may be influenced by the requirements of the application.

Following sections of the chapter deal with a number of loosely-related topics that are of importance to programmers working in the application development environment. Most of these mirror topics that were discussed in Chapter 1, but here we try to point out aspects of the subject that are particularly pertinent to application programming. They are covered under the following headings:

Advanced Programming Tools	Deals with such topics as File and Record Locking, Interprocess Communication, and programming terminal screens.
Programming Support Tools	Covers the Common Object File Format, link editor directives, shared libraries, **pdbx**, and **lint**.
Project Control Tools	Includes some discussion of **make** and SCCS.

The chapter concludes with a description of a sample application called **liber** that uses several of the components described in earlier portions of the chapter.

2.2 Application Programming

The characteristics of the application programming environment that make it different from single-user programming have at their base the need for interaction and for sharing of information.

2.2.1 Numbers

Perhaps the most obvious difference between application programming and single-user programming is in the quantities of the components. Not only are applications generally developed by teams of programmers, but the number of separate modules of code can grow into the hundreds on even a fairly simple application.

When more than one programmer works on a project, there is a need to share information such as the following:

- The operation of each function
- The number, identity and type of arguments expected by a function
- If pointers are passed to a function, let somebody know if the objects being pointed to modified by the called function, and what the lifetime of the pointed-to object is
- The data type returned by a function

In an application, there is a good chance that the same function can be used in many different programs, by many different programmers. The object code needs to be kept in a library accessible to anyone on the project who needs it.

2.2.2 Portability

When you are working on a program to be used on a single model of a computer, your concerns about portability are minimal. In application development, on the other hand, a desirable objective often is to produce code that will run on many different computers. Some of the things that affect portability will be discussed later in this chapter.

2.2.3 Documentation

A single-user program has modest needs for documentation. There should be enough present to remind the program's creator how to use it and, in portions of the code, enough to explain what the intent was.

On an application development project there is a significant need for two types of internal documentation:

- Comments throughout the source code that enable successive programmers to understand easily what is happening in the code. Applications can be expected to have a useful life of five or more years, and frequently need to be modified during that time. It is not realistic to expect that the same person who wrote the program will always be available to make modifications. Even if that does happen, the comments will make the maintenance job a lot easier.
- Hard-copy descriptions of functions should be available to all members of an application development team. Without them it is difficult to keep track of available modules, which can result in the same function being written over again.

Unless end-users have clear, readily-available instructions in how to install and use an application, they either will not do it at all (if that is an option), or do it improperly.

The microcomputer software industry has become ever more keenly aware of the importance of good end-user documentation. There are cases on record where the success of a software package has been attributed in large part to the fact that it had exceptionally good documentation. There are also cases where a pretty good piece of software was not widely used due to the inaccessibility of its manuals.

2.3 Language Selection

In this section we talk about some of the considerations that influence the selection of programming languages, and describe two of the special-purpose languages that are part of the system environment.

2.3.1 Influences

In single-user programming the choice of language is often a matter of personal preference; a language is chosen because it is the one the programmer feels most comfortable with.

An additional set of considerations comes into play when making the same decision for an application development project.

- Is there an existing standard within the organization that should be observed? A firm may decide to emphasize one language because a good supply of programmers is available who are familiar with it.
- Does one language have better facilities for handling the particular algorithm? One would like to see all language selections based on such objective criteria, but it is often necessary to balance this against the skills of the organization.
- Is there an inherent compatibility between the language and the operating system? This is sometimes the impetus behind selecting C for programs destined for a system machine.
- Are there existing tools that can be used? If parsing of input lines is an important phase of the application, perhaps a parser generator such as **yacc** should be employed to develop what the application needs.
- Does the application integrate other software into the whole package? If, for example, a package is to be built around an existing database management system, there may be constraints on the variety of languages the database management system can accommodate.

2.3.2 Special-Purpose Languages

The system contains a number of tools that can be included in the category of special-purpose languages. Three that are especially interesting are **awk**, **lex**, and **yacc**.

What awk Is Like

The **awk** utility scans an ASCII input file record-by-record, looking for matches to specific patterns. When a match is found, an action is taken. Patterns and their accompanying actions are contained in a specification file referred to as the *program*. The program can be made up of a number of statements. However, since each statement has the potential for causing a complex action, most **awk** programs consist of only a few. The set of

statements may include definitions of the pattern that separates one record from another (a newline character, for example), and what separates one field of a record from the next (white space, for example). It may also include actions to be performed before the first record of the input file is read and other actions to be performed after the final record has been read. All statements in between are evaluated in order for each record in the input file. To paraphrase the action of a simple **awk** program, it would go something like this:

Look through the input file.
Every time you see this specific pattern, do this action.

A more complex **awk** program might be paraphrased like this:

First, do some initialization.
Then, look through the input file.
Every time you see this specific pattern, do this action.
Every time you see this other pattern, do another action.
After all the records have been read, do these final things.

The directions for finding the patterns and for describing the actions can get pretty complicated, but the essential idea is as simple as the two sets of statements above.

One of the strong points of **awk** is that once you are familiar with the language syntax, programs can be written very quickly. They don't always run very fast, however, so they are seldom appropriate if you want to run the same program repeatedly on large quantities of records. In such a case, it is likely to be better to translate the program to a compiled language.

How awk Is Used

One typical use of **awk** would be to extract information from a file and print it out in a report. Another might be to pull fields from records in an input file, arrange them in a different order, and pass the resulting rearranged data to a function that adds records to your database. There is an example of a use of **awk** in the sample application at the end of this chapter.

Other Information About awk

The manual page for **awk** is in section (1) of the *Reference Manual.* The *Programming Tools Guide* contains a description of the **awk** syntax and a number of examples showing ways in which **awk** may be used.

What lex and yacc Are Like

lex and **yacc** are often mentioned in the same breath because they perform complementary parts of what can be viewed as a single task: making sense out of formal input. The two utilities also share the common characteristic of producing source code for C language subroutines from specifications that appear on the surface to be quite similar.

Recognizing input is a recurring problem in programming. Input can be from various sources. In a language compiler, for example, the input is normally contained in a file of source language statements. The system shell language most often receives its input from a person keying in commands from a terminal. Frequently, information coming out of one program is fed into another where it must be evaluated.

The process of input recognition can be subdivided into two tasks: lexical analysis and parsing—that's where **lex** and **yacc** come in. In both utilities, the specifications cause the generation of C language subroutines that deal with streams of characters; **lex** generates subroutines that do lexical analysis while **yacc** generates subroutines that do parsing.

To describe those two tasks in dictionary terms:

> *Lexical analysis* has to do with identifying the words or vocabulary of a language as distinguished from its grammar or structure.
>
> *Parsing* is the act of describing units of the language grammatically. Students in elementary school are often taught to do this with sentence diagrams.

The important thing to remember here is that in each case the rules for our lexical analysis or parsing are those we set down ourselves in the **lex** or **yacc** specifications. Because of this, the dividing line between lexical analysis and parsing sometimes becomes fuzzy.

The fact that **lex** and **yacc** produce C language source code means that these parts of what may be a large programming project can be separately maintained. The generated source code is processed by the C compiler to produce an object file. The object file can be link-edited with others to produce programs that then perform whatever process follows from the recognition of the input.

How lex Is Used

A **lex** subroutine scans a stream of input characters and waves a flag each time it identifies something that matches one or another of its rules. The waved flag is referred to as a token. The rules are stated in a format that closely resembles the one used by the system text editor for regular expressions. For example,

```
[ \t]+
```

describes a rule that recognizes a string of one or more blanks or tabs (without mentioning any action to be taken). A more complete statement of that rule might have this notation:

```
[ \t]+ ;
```

which, in effect, says to ignore white space. It carries this meaning because no action is specified when a string of one or more blanks or tabs is recognized. The semicolon marks the end of the statement. Another rule, one that does take some action, could be stated like this:

```
[0-9]+ {
    i = atoi(yytext);
    return(NBR);
    }
```

This rule depends on several things:

- NBR must have been defined as a token in an earlier part of the **lex** source code called the declaration section. (It may be in a header file which is included in the declaration section.)
- **i** is declared as an **extern int** in the declaration section.
- It is a characteristic of **lex** that things it finds are made available in a character string called **yytext**.
- Actions can make use of standard C syntax. Here, the standard C subroutine, **atoi**, is used to convert the string to an integer.

What this rule boils down to is that **lex** finds the kind of token specifically looked for and tells you where the value is stored.

To review the steps of the process:

1. The **lex** specification statements are processed by the **lex** utility to produce a file called **lex.yy.c**. (This is the standard name for a file generated by **lex**, just as **a.out** is the standard name for the executable file generated by the link editor.)
2. **lex.yy.c** is transformed by the C compiler (with a **–c** option) into an object file called **lex.yy.o** that contains a subroutine called **yylex()**.
3. **lex.yy.o** is link-edited with other subroutines. Presumably one of those subroutines will call **yylex()** with a statement such as:

```
while((token = yylex()) != 0)
```

   and other subroutines (or even **main**) will deal with what comes back.

Other Information About lex

The manual page for **lex** is in Section (1) of the *Reference Manual.* A good discussion on **lex** is contained in the *Programming Tools Guide.*

How yacc Is Used

yacc subroutines are produced by the same basic series of steps as **lex**:

1. The **yacc** specification is processed by the **yacc** utility to produce a file called **y.tab.c**.
2. **y.tab.c** is compiled by the C compiler producing an object file, **y.tab.o**, that contains the subroutine **yyparse()**. A significant difference is that **yyparse()** calls a subroutine called **yylex()** to perform lexical analysis.
3. The object file **y.tab.o** may be link-edited with other subroutines, one of which will be called **yylex()**.

There are two things worth noting about this sequence:

1. The parser generated by the **yacc** specifications calls a lexical analyzer to scan the input stream and return tokens.
2. While the lexical analyzer is called by the same name as one produced by **lex**, it does not have to be the product of a **lex** specification. It can be any subroutine that does the lexical analysis.

What really differentiates these two utilities is the format for their rules. As noted above, **lex** rules are regular expressions like those used by system editors. **yacc** rules are chains of definitions and alternative definitions, written in Backus-Naur form, accompanied by actions. The rules may refer to other rules defined later in the specification. Actions are sequences of C language statements enclosed in braces. They frequently contain numbered variables that enable you to reference values associated with parts of the rules. Consider the next example. It might make this easier to understand:

```
%token      NUMBER
%%
expr   : numb                       { $$ = $1; }
       | expr '+' expr              { $$ = $1 + $3; }
       | expr '-' expr              { $$ = $1 - $3; }
       | expr '*' expr              { $$ = $1 * $3; }
       | expr '/' expr              { $$ = $1 / $3; }
       | '(' expr ')'               { $$ = $2; }
       ;
numb   : NUMBER            { $$ = $1; }
       ;
```

This fragment of a **yacc** specification shows:

- NUMBER identified as a token in the declaration section
- The start of the rules section indicated by the pair of percent signs
- A number of alternate definitions for *expr* separated by the | sign and terminated by the semicolon
- Actions to be taken when a rule is matched
- Within actions, numbered variables used to represent components of the rule:

  $$ means the value to be returned as the value of the whole rule

  $*n* means the value associated with the *n*th component of the rule, counting from the left

- *numb* defined as meaning the token NUMBER. This is a trivial example that illustrates that one rule can be referenced within another, as well as within itself.

As with **lex**, the compiled **yacc** object file will generally be link-edited with

other subroutines that handle processing that takes place after the parsing—or even ahead of it.

Other Information About yacc

The manual page for **yacc** is in Section (1) of the *Reference Manual.* A detailed description of **yacc** may be found in the *Programming Tools Guide.*

2.4 Advanced Programming Tools

In Chapter 1 we described the use of such basic elements of programming in the system environment as the standard I/O library, header files, system calls, and subroutines. In this section we introduce tools that are more apt to be used by members of an application development team than by a single-user programmer. The section contains material on the following topics:

- Memory management
- File and record locking
- Interprocess communication
- Terminal screen programming

2.4.1 Memory Management

There are situations where a program needs to ask the operating system for blocks of memory. It may be, for example, that a number of records have been extracted from a database and need to be held for some further processing. Rather than writing them out to a file on secondary storage and then reading them back in again, it will likely be more efficient to hold them in memory for the duration of the process. (This is not to ignore the possibility that portions of memory may be paged out before the program is finished, but such an occurrence is not pertinent to this discussion.)

There are two C language subroutines available for acquiring blocks of memory and they are both called **malloc**: **malloc**(3C) and **malloc**(3X). Each has several related commands that do specialized tasks in the same area. They are as follows:

- **free**—to inform the system that space is being relinquished
- **realloc**—to change the size and possibly move the block
- **calloc**—to allocate space for an array and initialize it to zeros

In addition, **malloc**(3X) has a function, **mallopt**, that provides for control over the space allocation algorithm, and a structure, **mallinfo**, from which the program can get information about the usage of the allocated space.

malloc(3X) runs faster than the other version. It is loaded by specifying

-lmalloc

on the **cc**(1) or **ld**(1) command line to direct the link editor to the proper library. When you use **malloc**(3X), your program should contain the following statement:

```
#include <malloc.h>
```

This points to where the values for **mallopt** options are defined.

See the *Reference Manual* for the formal definitions of the two **malloc** subroutines.

2.4.2 File and Record Locking

The provision for locking files, or portions of files, is primarily used to prevent the sort of error that can occur when two or more users of a file try to update information at the same time. The classic example is the airlines reservation system where two ticket agents each assign a passenger to Seat A, Row 5 on the 5 o'clock flight to Detroit. A locking mechanism is designed to prevent such mishaps by blocking Agent B from even seeing the seat assignment file until Agent A's transaction is complete.

File locking and record locking are really the same thing, except that file locking implies that the whole file is affected; record locking means that only a specified portion of the file is locked. (Remember that, in the operating system, file structure is undefined; a record is a concept of the programs that use the file.)

Two types of locks are available: read locks and write locks. If a process places a read lock on a file, other processes can also read the file but all are prevented from writing to it (that is, changing any of the data). If a process places a write lock on a file, no other processes can read or write in the file until the lock is removed. Write locks are also known as exclusive locks. The term *shared lock* is sometimes applied to read locks.

Another distinction needs to be made between mandatory and advisory locking. *Mandatory locking* means that the discipline is enforced automatically for the system calls that read, write, or create files. This is done through a permission flag established by the file's owner (or the superuser). *Advisory locking* means that the processes that use the file take the responsibility for setting and removing locks as needed. The mandatory locking capability is included in the system to comply with an agreement with */usr/group*, an organization that represents the interests of system users. The principal weakness in the mandatory method is that the lock is in place only while the single system call is being made. It is extremely common for a single transaction to require a series of reads and writes before it can be considered complete. In cases like this, the term atomic is used to describe a transaction that must be viewed as an indivisible unit. The preferred way to manage locking in such a circumstance is to make certain the lock is in place before any I/O starts, and that it is not removed until the transaction is done. That calls for locking of the advisory variety.

How File and Record Locking Works

The system call for file and record locking is **fcntl**(2). Programs should include the following line:

```
#include <fcntl.h>
```

This brings in the header file shown in the next example.

```
/* Flag values accessible to open(2) and fcntl(2) */
/*  (The first three can only be set by open) */
#define  O_RDONLY  0
#define  O_WRONLY  1
#define  O_RDWR    2
#define  O_NDELAY  04      /* Non-blocking I/O */
#define  O_APPEND  010     /* append (writes guaranteed at the end) */
#define  O_SYNC            020/* synchronous write option */
/* Flag values accessible only to open(2) */
#define  O_CREAT   00400   /* open with file create (uses third open arg)*/
#define  O_TRUNC   01000   /* open with truncation */
#define  O_EXCL    02000   /* exclusive open */
/* fcntl(2) requests */
#define  F_DUPFD   0       /* Duplicate fildes */
#define  F_GETFD   1       /* Get fildes flags */
#define  F_SETFD   2       /* Set fildes flags */
#define  F_GETFL   3       /* Get file flags */
#define  F_SETFL   4       /* Set file flags */
```

```
#define  F_GETLK   5       /* Get file lock */
#define  F_SETLK   6       /* Set file lock */
#define  F_SETLKW  7       /* Set file lock and wait */
#define  F_CHKFL   8       /* Check legality of file flag changes */

/* file segment locking set data type - info passed to system by user */
struct flock {
         short     l_type;
         short     l_whence;
         long      l_start;
         long      l_len;  /* len = 0 means until end of file */
         short     l_sysid;
         short     l_pid;
};

/* file segment locking types */
         /* Read lock */
#define  F_RDLCK   01
         /* Write lock */
#define  F_WRLCK   02
         /* Remove lock(s) */
#define  F_UNLCK   03
```

The format of the **fcntl**(2) system call is as follows:

```
int fcntl(fildes, cmd, arg)
int fildes, cmd, arg;
```

fildes is the file descriptor returned by the **open** system call. In addition to defining tags that are used as the commands on **fcntl** system calls, **fcntl.h** includes the declaration for a *struct flock* that is used to pass values that control where locks are to be placed.

lockf

A subroutine, **lockf**(3), can also be used to lock sections of a file or an entire file. The format of **lockf** is:

```
#include <unistd.h>

int lockf (fildes, function, size)
int fildes, function;
long size;
```

fildes is the file descriptor; *function* is one of four control values defined in **unistd.h** that let you lock, unlock, test and lock, or simply test to see if a lock is already in place. *size* is the number of contiguous bytes to be locked or unlocked. The section of contiguous bytes can be either forward or backward from the current offset in the file. (You can arrange to be somewhere in the middle of the file by using the **lseek**(2) system call.)

Other Information

There is an example of file and record locking in the sample application at the end of this chapter. The manual pages that apply to this facility are **fcntl**(2), **fcntl**(5), **lockf**(3), and **chmod**(2) in the *Reference Manual.* The *Programming Tools Guide* has a detailed discussion of the subject with a number of examples.

2.4.3 Interprocess Communications

In Chapter 1 we described **fork**ing and **exec**ing as methods of communicating between processes. Business applications running on a computer often need more sophisticated methods. In applications, for example, where fast response is critical, a number of processes may be brought up at the start of a business day to be constantly available to handle transactions on demand. This cuts out initialization time that can add seconds to the time required to deal with the transaction. Returning to the ticket reservation example, for a moment, if a customer calls to reserve a seat on the 5 o'clock flight to Detroit, you don't want to have to say, "Yes, sir. Just hang on a minute while I start up the reservations program." In transaction-driven systems, the normal mode of processing is to have all the components of the application standing by waiting for some sort of an indication that there is work to do.

To meet requirements of this type, the operating system offers a set of nine system calls and their accompanying header files, all under the umbrella name of Interprocess Communications (IPC).

The IPC system calls come in sets of three; one set each for messages, semaphores, and shared memory. These three terms define three different styles of communication between processes:

Messages	Communication is in the form of data stored in a buffer. The buffer can be either sent or received.

Semaphores	Communication is in the form of positive integers with a value between 0 and 32,767. Semaphores may be contained in an array the size of which is determined by the system administrator. The default maximum size for the array is 25.
Shared memory	Communication takes place through a common area of main memory. One or more processes can attach a segment of memory and as a consequence can share whatever data is placed there.

The sets of IPC system calls are:

msgget	**semget**	**shmget**
msgctl	**semctl**	**shmctl**
msgop	**semop**	**shmop**

IPC get Calls

The **get** calls each return to the calling program an identifier for the type of IPC facility that is being requested.

IPC ctl Calls

The **ctl** calls provide a variety of control operations that include obtaining (IPC_STAT), setting (IPC_SET) and removing (IPC_RMID), the values in data structures associated with the identifiers picked up by the **get** calls.

IPC op Calls

The **op** manual pages describe calls that are used to perform the particular operations characteristic of the type of IPC facility being used. **msgop** has calls that send or receive messages. **semop** (the only one of the three that is actually the name of a system call) is used to increment or decrement the value of a semaphore, among other functions. **shmop** has calls that attach or detach shared memory segments.

Other Information

An example of the use of some IPC features is included in the sample application at the end of this chapter. The system calls are all located in Section 2 of the *Reference Manual.* Don't overlook **intro**(2). It includes descriptions of the data structures that are used by IPC facilities. A detailed description of IPC, with many code examples that use the IPC system calls, is contained in Chapter 4 in this guide.

2.4.4 Programming Terminal Screens

The facility for setting up terminal screens to meet the needs of your application is provided by two parts of the operating system. The first of these, **terminfo**, is a database of compiled entries that describe the capabilities of terminals and the way they perform various operations.

The **terminfo** database normally begins at the directory */usr/lib/terminfo*. The members of this directory are themselves directories, generally with single-character names that are the first character in the name of the terminal. The compiled files of operating characteristics are at the next level down the hierarchy. For example, the entry for a Teletype 5425 is located in both the file */usr/lib/terminfo/5/5425* and the file */usr/lib/terminfo/t/tty5425*.

Describing the capabilities of a terminal can be a painstaking task. Quite a good selection of terminal entries is included in the **terminfo** database that comes with your computer. However, if you have a type of terminal that is not already described in the database, the best way to proceed is to find a description of one that comes close to having the same capabilities as yours and building on that one. There is a routine (**setupterm**) in **curses**(3X) that can be used to print out descriptions from the database. Once you have worked out the code that describes the capabilities of your terminal, the **tic**(1M) command is used to compile the entry and add it to the database.

curses

After you have made sure that the operating capabilities of your terminal are a part of the **terminfo** database, you can then proceed to use the routines that make up the **curses**(3X) package to create and manage screens for your application.

The **curses** library includes functions to handle the following tasks:

- Define portions of your terminal screen as windows.
- Define pads that extend beyond the borders of your physical terminal screen and let you see portions of the pad on your terminal.
- Read input from a terminal screen into a program.
- Write output from a program to your terminal screen.
- Manipulate the information in a window in a virtual screen area and then send it to your physical screen.

Other Information About curses and terminfo

In the sample application at the end of this chapter, we show how you might use **curses** routines. Chapter 5 of this guide is devoted to curses/terminfo. The manual pages for **curses** are in Section (3X) and those for **terminfo** are in Section (4) of the *Reference Manual.*

2.5 Programming Support Tools

This section covers system components that are part of the programming environment, but that have a highly specialized use. These components include the following:

- Link edit command language
- Common Object File Format
- Libraries
- Symbolic Debugger
- **lint** as a portability tool

2.5.1 Link Edit Command Language

The link editor command language is for use when the default arrangement of the **ld** output will not do the job. The default locations for the standard Common Object File Format sections are described in **a.out**(4) in the *Reference Manual.* When an **a.out** file is loaded into memory for execution, the text segment starts at location 0x80800000, and the data section starts at the next segment boundary after the end of the text. The stack begins at 0xC0020000 and grows to higher memory addresses.

The link editor command language provides directives for describing different arrangements. The two major types of link editor directives are MEMORY and SECTIONS. MEMORY directives can be used to define the boundaries of configured and unconfigured sections of memory within a machine, to name sections, and to assign specific attributes (read, write, execute, and initialize) to portions of memory. SECTIONS directives, among many functions, can be used to bind sections of the object file to specific addresses within the configured portions of memory.

Why would you want to be able to do those things? Actually, in the majority of cases you don't have to worry about it. The need to control the link editor output becomes more urgent under two sets of possibly related circumstances:

1. Your application is large and consists of a lot of object files.
2. The hardware your application is to run on is tight for space.

Other Information

The *Programming Tools Guide* gives a detailed description of the subject.

2.5.2 Common Object File Format

A knowledge of COFF is fundamental to using the link editor command language. It is also good background knowledge for tasks such as:

- Setting up archive libraries or shared libraries
- Using the Symbolic Debugger

The following system header files contain definitions of data structures of parts of the Common Object File Format:

<syms.h>	Symbol table format
<linenum.h>	Line number entries
<ldfcn.h>	COFF access routines
<filehdr.h>	File header for a common object file
<a.out.h>	Common assembler and link editor output
<scnhdr.h>	Section header for a common object file
<reloc.h>	Relocation information for a common object file
<storclass.h>	Storage classes for common object files

The object file access routines are described below.

Other Information

Chapter 6 of this guide gives a detailed description of COFF.

2.5.3 Libraries

A library is a collection of related object files and declarations that simplify the programming effort. Programming groups involved in the development of applications often find it convenient to establish private libraries. For example, an application with a number of programs using a common database can keep the I/O routines in a library that is searched at link-edit time.

In Chapter 2 we described many of the functions that are found in the standard C library, **libc.a**. The next two sections describe two other libraries, the object file library and the math library.

The Object File Library

The object file library provides functions for the access and manipulation of object files. Some functions locate portions of an object file such as the symbol table, the file header, sections, and line number entries associated with a function. Other functions read these types of entries into memory. The need to work at this level of detail with object files occurs most often in the development of new tools that manipulate object files. For a description of the format of an object file, refer to "The Common Object File Format" in Chapter 6. This library consists of several portions. The functions reside in **/lib/libld.a** and are loaded during the compilation of a C language program by the *–l* command line option:

```
cc file -lld
```

which causes the link editor to search the object file library. The argument

–lld must appear after all files that reference functions in **libld.a**.

The following header files must be included in the source code.

```
#include <stdio.h>
#include <a.out.h>
#include <ldfcn.h>
```

Common Object File Interface Macros (ldfcn.h)

Table 2-1 shows the object file interface macros.

Table 2-1
Object File Interface Macros

Function	Reference	Brief Description
ldaclose	**ldclose**(3X)	Close object file being processed.
ldahread	**ldahread**(3X)	Read archive header.
ldaopen	**ldopen**(3X)	Open object file for reading.
ldclose	**ldclose**(3X)	Close object file being processed.
ldfhread	**ldfhread**(3X)	Read file header of object file being processed.
ldgetname	**ldgetname**(3X)	Retrieve the name of an object file symbol table entry.
ldlinit	**ldlread**(3X)	Prepare object file for reading line number entries via **ldlitem**.
ldlitem	**ldlread**(3X)	Read line number entry from object file after **ldlinit**.
ldlread	**ldlread**(3X)	Read line number entry from object file.
ldlseek	**ldlseek**(3X)	Seeks to the line number entries of the object file being processed.
ldnlseek	**ldlseek**(3X)	Seeks to the line number entries of the object file being processed given the name of a section.

Table 2-1
Object File Interface Macros (cont.)

Function	Reference	Brief Description
ldnrseek	**ldrseek**(3X)	Seeks to the relocation entries of the object file being processed given the name of a section.
ldnshread	**ldshread**(3X)	Read section header of the named section of the object file being processed.
ldnsseek	**ldsseek**(3X)	Seeks to the section of the object file being processed given the name of a section.
ldohseek	**ldohseek**(3X)	Seeks to the optional file header of the object file being processed.
ldopen	**ldopen**(3X)	Open object file for reading.
ldrseek	**ldrseek**(3X)	Seeks to the relocation entries of the object file being processed.
ldshread	**ldshread**(3X)	Read section header of an object file being processed.
ldsseek	**ldsseek**(3X)	Seeks to the section of the object file being processed.

Table 2-1
Object File Interface Macros (cont.)

Function	Reference	Brief Description
ldtbindex	**ldtbindex**(3X)	Returns the long index of the symbol table entry at the current position of the object file being processed.
ldtbread	**ldtbread**(3X)	Reads a specific symbol table entry of the object file being processed.
ldtbseek	**ldtbseek**(3X)	Seeks to the symbol table of the object file being processed.
sgetl	**sputl**(3X)	Access long integer data in a machine-independent format.
sputl	**sputl**(3X)	Translate a long integer into a machine-independent format.

The interface between the calling program and the object file access routines is based on the defined type LDFILE, which is defined in the header file **ldfcn.h** (see **ldfcn**(4)). The primary purpose of this structure is to provide uniform access to both simple object files and to object files that are members of an archive file.

The function **ldopen**(3X) allocates and initializes the LDFILE structure and returns a pointer to the structure. The fields of the LDFILE structure may be accessed individually through the following macros:

- The TYPE macro returns the magic number of the file, which is used to distinguish between archive files and object files that are not part of an archive.
- The IOPTR macro returns the file pointer, which was opened by **ldopen**(3X) and is used by the input/output functions of the C library.
- The OFFSET macro returns the file address of the beginning of the object file. This value is nonzero only if the object file is a member of the archive file.

- The HEADER macro accesses the file header structure of the object file.

Additional macros are provided to access an object file. These macros parallel the input/output functions in the C library; each macro translates a reference to an LDFILE structure into a reference to its file descriptor field. The available macros are described in **ldfcn**(4) in the *Reference Manual.*

The Math Library

The math library package consists of functions and a header file. The functions are located and loaded during the compilation of a C language program by the –l option on a command line, as follows:

cc *file* **–lm**

This option causes the link editor to search the math library, **libm.a**. In addition to the request to load the functions, the header file of the math library should be included in the program being compiled. This is accomplished by including the line near the beginning of each file that uses the routines:

```
#include <math.h>
```

The functions are grouped into the following categories:

- Trigonometric functions
- Bessel functions
- Hyperbolic functions
- Miscellaneous functions

2.5.4 Trigonometric Functions

The functions shown in table 2-2 are used to compute angles (in radian measure), sines, cosines, and tangents. All of these values are expressed in double-precision format.

Table 2-2
Trigonometric Functions

Function	Reference	Brief Description
acos	**trig**(3M)	Return arc cosine.
asin	**trig**(3M)	Return arc sine.
atan	**trig**(3M)	Return arc tangent.
atan2	**trig**(3M)	Return arc tangent of a ratio.
cos	**trig**(3M)	Return cosine.
sin	**trig**(3M)	Return sine.
tan	**trig**(3M)	Return tangent.

Bessel Functions. These functions calculate Bessel functions of the first and second kinds of several orders for real values. The Bessel functions are **j0, j1, jn, y0, y1,** and **yn.** The functions are described in section **bessel**(3M).

Hyperbolic Functions. The functions shown in Table 2-3 are used to compute the hyperbolic sine, cosine, and tangent for real values.

Table 2-3
Hyperbolic Functions

Function	Reference	Brief Description
cosh	**sinh**(3M)	Return hyperbolic cosine.
sinh	**sinh**(3M)	Return hyperbolic sine.
tanh	**sinh**(3M)	Return hyperbolic tangent.

Miscellaneous Functions. The functions shown in Table 2-4 cover a wide variety of operations, such as natural logarithm, exponential, and absolute value. In addition, several are provided to truncate the integer portion of double-precision numbers.

Table 2-4
Miscellaneous Functions

Function	Reference	Brief Description
ceil	**floor**(3M)	Returns the smallest integer not less than a given value.
exp	**exp**(3M)	Returns the exponential function of a given value.
fabs	**floor**(3M)	Returns the absolute value of a given value.
floor	**floor**(3M)	Returns the largest integer not greater than a given value.
fmod	**floor**(3M)	Returns the remainder produced by the division of two given values.
gamma	**gamma**(3M)	Returns the natural log of the absolute value of the result of applying the gamma function to a given value.
hypot	**hypot**(3M)	Return the square root of the sum of the squares of two numbers.
log	**exp**(3M)	Returns the natural logarithm of a given value.
log10	**exp**(3M)	Returns the logarithm base ten of a given value.
matherr	**matherr**(3M)	Error-handling function.
pow	**exp**(3M)	Returns the result of a given value raised to another given value.
sqrt	**exp**(3M)	Returns the square root of a given value.

Shared Libraries

Not only are some system libraries (**libc** and the networking library) available in both archive and shared library form, but applications also have the option of creating private-application shared libraries.

The reason why shared libraries are desirable is that they save space, both on disk and in memory. With an archive library, when the link editor goes to the archive to resolve a reference, it takes a copy of the object file that it needs for the resolution and binds it into the **a.out** file. From that point on, the copied file is a part of the executable, whether it is in memory to be run or sitting in secondary storage. If you have a lot of executables that use, say, **printf** (which just happens to require much of the standard I/O library), you can be talking about a sizeable amount of space.

With a shared library, the link editor does not copy code into the executable files. When the operating system starts a process that uses a shared library, it maps the shared library contents into the address space of the process. Only one copy of the shared code exists, and many processes can use it at the same time.

This fundamental difference between archives and shared libraries has another significant aspect. When code in an archive library is modified, all existing executables are unaffected. They continue using the older version until they are re-link-edited. When code in a shared library is modified, all programs that share that code use the new version the next time they are executed.

All this may sound really terrific, but as with most things in life there are complications. To begin with, in the paragraphs above you didn't quite get all the facts. For example, each process that uses shared library code gets its own copy of the entire data region of the library. It is actually only the text region that is really shared. So the truth is that shared libraries can add space to executing **a.out**'s even though the chances are good that they will cause more shrinkage than expansion. What this means is that when there is a choice between using a shared library and an archive, you shouldn't use the shared library unless it saves space. If you were using a shared **libc** to access only **strcmp**, for example, you would pick up more in shared library data than you would save by sharing the text.

The answer to this problem, and to others that are somewhat more complex, is to assign the responsibility for shared libraries to a central person or group within the application. The shared library developer should be the one to resolve questions of when to use shared and when to use archive system libraries. If a private library is to be built for your application, one person or

organization should be responsible for its development and maintenance.

Other Information About Shared Libraries

The sample application at the end of this chapter includes an example of the use of a shared library. The *Programming Tools Guide* describes how shared libraries are built and maintained.

2.5.5 Pdbx — A Symbolic Debugger

Pdbx is an interactive, symbolic debugger for use in debugging applications written in C, FORTRAN, Pascal, assembly language, or any combination of these languages. You can use Pdbx to debug both conventional, one-process applications and parallel applications (those consisting of two or more cooperating processes executing concurrently). You can also use Pdbx to debug applications that consist of multiple processes running different programs such as client/server applications.

Other Information About Pdbx

More information about **Pdbx** is found in the *Pdbx User's Guide*. The manual page is in the Language Tools binder.

2.5.6 lint as a Portability Tool

It is a characteristic of the operating system that language compilation systems are somewhat permissive. Generally speaking, it is a design objective that a compiler should run fast. Most C compilers, therefore, let some things go unflagged as long as the language syntax is observed statement-by-statement. This sometimes means that while your program may run, the output will have some surprises. It also sometimes means that while the program may run on the machine on which the compilation system runs, there may be real difficulties in running it on some other machine.

That's where **lint** comes in. **lint** produces comments about inconsistencies in the code. The types of anomalies flagged by **lint** are:

- Cases of disagreement between the type of value expected from a called function and what the function actually returns
- Disagreement between the types and number of arguments expected by functions and what the function receives

- Inconsistencies that might prove to be bugs
- Things that might cause portability problems

Here is an example of a portability problem that would be caught by **lint**.

Code such as the following would get by most compilers:

```
int i = lseek(fdes, offset, whence)
```

However, **lseek** returns a long integer representing the address of a location in the file. On a machine with a 16-bit integer and a bigger **long int**, it would produce incorrect results, because **i** would contain only the last 16 bits of the value returned.

Since it is reasonable to expect that an application written for a machine will be able to run on a variety of computers, it is important that the use of **lint** be a regular part of the application development.

Other Information

The *Programming Tools Guide* contains a description of **lint** with examples of the kinds of conditions it uncovers. The manual page is in Section (1) of the *Reference Manual.*

2.6 Project Control Tools

Project control is an item of top priority for the managers of any application development team. Two system tools that can play a role in this area are described in this section: **make** and SCCS.

2.6.1 make

make is extremely useful in an application development project for keeping track of what object files need to be recompiled as changes are made to source code files. One of the characteristics of programs in a system environment is that they are made up of many small pieces, each in its own object file, that are link-edited together to form the executable file. Quite a few of the system tools are devoted to supporting that style of program architecture. For example, archive libraries, shared libraries and even the fact that the **cc** command accepts **.o** files as well as **.c** files, and that it can stop short of the **ld** step and produce **.o** files instead of an **a.out**, are all important elements of modular architecture. The two main advantages of this type of programming are as follows:

- A file that performs one function can be re-used in any program that needs it.
- When one function is changed, the whole program does not have to be recompiled.

On the flip side, however, a consequence of the proliferation of object files is an increased difficulty in keeping track of what does need to be recompiled and what doesn't. **make** is designed to help deal with this problem. You use **make** by describing in a specification file, called **makefile**, the relationship (that is, the dependencies) between the different files of your program. Once having done that, you conclude a session in which a number of your source code files have been changed by running the **make** command. **make** takes care of generating a new **a.out** by comparing the time-last-changed of your source code files with the dependency rules you have given it.

make has the ability to work with files in archive libraries or under control of the Source Code Control System (SCCS).

Other Information About make

The **make**(1) manual page is contained in the *Reference Manual.* The *Programming Tools Guide* gives a complete description of how to use **make**.

2.6.2 SCCS

SCCS is an acronym for Source Code Control System. It consists of a set of 14 commands used to track evolving versions of files. Its use is not limited to source code; any text files can be handled, so an application's documentation can also be put under control of SCCS. SCCS can perform the following tasks:

- Store and retrieve files under its control
- Allow no more than a single copy of a file to be edited at one time
- Provide an audit trail of changes to files
- Reconstruct any earlier version of a file that may be wanted

SCCS files are stored in a special coded format. Only through commands that are part of the SCCS package can files be made available in a user's directory for editing, compiling, and so on. From the point at which a file is first placed under SCCS control, only changes to the original version are stored. For example, let's say that the program, **restate**, that was used in

several examples in Chapter 2, was controlled by SCCS. One of the original pieces of that program is a file called **oppty.c** that looks like this:

```
        /* Opportunity Cost -- oppty.c */
#include "recdef.h"

float
oppty(ps)
struct rec *ps;
{
    return(ps->i/12 * ps->t * ps->dp);
}
```

If you decide to add a message to this function, you might change the file like this:

```
        /* Opportunity Cost -- oppty.c */
#include "recdef.h"
#include <stdio.h>

float
oppty(ps)
struct rec *ps;
{
    (void) fprintf(stderr, "Opportunity calling\n");
    return(ps->i/12 * ps->t * ps->dp);
}
```

SCCS saves only the two new lines from the second version, with a coded notation that shows where in the text the two lines belong. It also includes a note of the version number, lines deleted, lines inserted, total lines in the file, the date and time of the change, and the login id of the person making the change.

Other Information

The *Programming Tools Guide* contains an SCCS user's guide. SCCS commands are in Section (1) of the *Reference Manual.*

2.7 Example — liber, A Library System

To illustrate the use of system programming tools in the development of an application, imagine that we are engaged in the development of a computer system for a library. The system is known as **liber**. The early stages of system development, we assume, have already been completed; feasibility studies have been done, the preliminary design is described in the coming paragraphs. We are going to stop short of producing a complete detailed design and module specifications for our system. You will have to accept that these exist. In using portions of the system for examples of the topics covered in this chapter, we will work from these virtual specifications.

We make no claim as to the efficiency of this design. It serves only to provide some passably realistic examples of system programming tools in use.

liber is a system for keeping track of the books in a library. The hardware consists of a single computer with terminals throughout the library. One terminal is used for adding new books to the database. Others are used for checking out books and as electronic card catalogs.

The design of the system calls for it to be brought up at the beginning of the day and to remain running while the library is in operation. The system has one master index that contains the unique identifier of each title in the library. When the system is running, the index resides in memory. Semaphores are used to control access to the index. In the pages that follow, fragments of some of the system's programs are shown to illustrate the way they work together. The start-up program performs the system initialization, opens the semaphores and shared memory, reads the index into the shared memory, and kicks off the other programs. The id numbers for the shared memory and semaphores (**shmid**, **wrtsem**, and **rdsem**) are read from a file during initialization. The programs all share the in-memory index. They attach it with the following code:

```
/* attach shared memory for index */
if ((int)(index = (INDEX *) shmat(shmid, NULL, 0)) == -1)
{
        (void) fprintf(stderr, "shmat failed: %d\n", errno);
        exit(1);
}
```

Of the programs shown, **add-books** is the only one that alters the index. The semaphores are used to ensure that no other programs will try to read the index while **add-books** is altering it. The checkout program locks the file record for the book, so that each copy being checked out is recorded separately and the book cannot be checked out at two different checkout stations at the same time.

The program fragments do not provide any details on the structure of the index or the book records in the database.

```
                /* liber.h - header file for the
                 *           library system.
                 */
typedef ... INDEX;/* data structure for book file index */
typedef struct {  /* type of records in book file */
    char title[30];
    char author[30];
    .
    .
    .
} BOOK;
int shmid;
int wrtsem;
int rdsem;
INDEX *index;

int book_file;
BOOK book_buf;
/*   startup program*/

/*
 * 1. Open shared memory for file index and read it in.
 * 2. Open two semaphores for providing exclusive write access to index.
 * 3. Stash id's for shared memory segment and semaphores in a file
 *    where they can be accessed by the programs.
 * 4. Start programs:  add-books, card-catalog, and checkout running
 *    on the various terminals throughout the library.
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

```
#include <sys/sem.h>
#include "liber.h"

void exit();
extern int errno;

key_t key;
int shmid;
int wrtsem;
int rdsem;
FILE *ipc_file;

main()
{
     .
     .
     .
     if ((shmid = shmget(key, sizeof(INDEX), IPC_CREAT | 0666)) == -1)
     {
          (void) fprintf(stderr, "startup: shmget failed: errno=%d\n", errno);
          exit(1);
     }
     if ((wrtsem = semget(key, 1, IPC_CREAT | 0666)) == -1)
     {
          (void) fprintf(stderr, "startup: semget failed: errno=%d\n", errno);
          exit(1);
     }
     if ((rdsem = semget(key, 1, IPC_CREAT | 0666)) == -1)
     {
          (void) fprintf(stderr, "startup: semget failed: errno=%d\n", errno);
          exit(1);
     }
     (void) fprintf(ipc_file, "%d\n%d\n%d\n", shmid, wrtsem, rdsem);

     /*
      * Start the add-books program running on the terminal in the
      * basement.  Start the checkout and card-catalog programs
      * running on the various other terminals throughout the library.
      */
     .
     .
     .
```

```
}

/*   card-catalog program*/

/*
 * 1. Read screen for author and title.
 * 2. Use semaphores to prevent reading index while it is being written.
 * 3. Use index to get position of book record in book file.
 * 4. Print book record on screen or indicate book was not found.
 * 5. Go to 1.
 */

#include        <stdio.h>
#include        <sys/types.h>
#include        <sys/ipc.h>
#include        <sys/sem.h>
#include <fcntl.h>
#include "liber.h"

void exit();
extern int errno;
struct sembuf sop[1];

main() {
     .
     .
     .
     while (1)
     {
          /*
           * Read author/title/subject information from screen.
           */

          /*
           * Wait for write semaphore to reach 0 (index not being written).
           */
          sop[0].sem_op = 1;
          if (semop(wrtsem, sop, 1) == -1)
          {
                    (void) fprintf(stderr, "semop failed: %d\n", errno);
                    exit(1);
          }
```

```
        /*
         * Increment read semaphore so potential writer will wait
         * for us to finish reading the index.
         */
        sop[0].sem_op = 0;
        if (semop(rdsem, sop, 1) == -1)
        {
                (void) fprintf(stderr, "semop failed: %d\n", errno);
                exit(1);
        }

        /* Use index to find file pointer(s) for book(s) */

        /* Decrement read semaphore */
        sop[0].sem_op = -1;
        if (semop(rdsem, sop, 1) == -1)
        {
                (void) fprintf(stderr, "semop failed: %d\n", errno);
                exit(1);
        }

        /*
         * Now we use the file pointers found in the index to
         * read the book file.  Then we print the information
         * on the book(s) to the screen.
         */
    } /* while */
}
/*   checkout program*/

/*
 * 1. Read screen for Dewey Decimal number of book to be checked out.
 * 2. Use semaphores to prevent reading index while it is being written.
 * 3. Use index to get position of book record in book file.
 * 4. If book not found print message on screen, otherwise lock
 *    book record and read.
 * 5. If book already checked out print message on screen, otherwise
 *    mark record "checked out" and write back to book file.
 * 6. Unlock book record.
 * 7. Go to 1.
 */
```

```
#include        <stdio.h>
#include        <sys/types.h>
#include        <sys/ipc.h>
#include        <sys/sem.h>
#include <fcntl.h>
#include "liber.h"

void exit();
long lseek();
extern int errno;
struct flock flk;
struct sembuf sop[1];
long bookpos;

main()
{
    .
    .
    .
    while (1)
    {
        /*
         * Read Dewey Decimal number from screen.
         */
        /*
         * Wait for write semaphore to reach 0 (index not being written).
         */
        sop[0].sem_flg = 0;
        sop[0].sem_op = 0;
        if (semop(wrtsem, sop, 1) == -1)
        {
                (void) fprintf(stderr, "semop failed: %d\n", errno);
                exit(1);
        }
        /*
         * Increment read semaphore so potential writer will wait
         * for us to finish reading the index.
         */
        sop[0].sem_op = 1;
        if (semop(rdsem, sop, 1) == -1)
        {
                (void) fprintf(stderr, "semop failed: %d\n", errno);
```

```
        exit(1);
}

/*
 * Now we can use the index to find the book's record position.
 * Assign this value to "bookpos".
 */

/* Decrement read semaphore */
sop[0].sem_op = -1;
if (semop(rdsem, sop, 1) == -1)
{
        (void) fprintf(stderr, "semop failed: %d\n", errno);
        exit(1);
}

/* Lock the book's record in book file, read the record. */
flk.l_type = F_WRLCK;
flk.l_whence = 0;
flk.l_start = bookpos;
flk.l_len = sizeof(BOOK);
if (fcntl(book_file, F_SETLKW, &flk) == -1)
{
        (void) fprintf(stderr, "trouble locking: %d\n", errno);
        exit(1);
}
if (lseek(book_file, bookpos, 0) == -1)
{
        Error processing for lseek;
}
if (read(book_file, &book_buf, sizeof(BOOK)) == -1)
{
        Error processing for read;
}

/*
 * If the book is checked out inform the client, otherwise
 * mark the book's record as checked out and write it
 * back into the book file.
 */

/* Unlock the book's record in book file. */
```

```
        flk.l_type = F_UNLCK;
        if (fcntl(book_file, F_SETLK, &flk) == -1)
        {
                (void) fprintf(stderr, "trouble unlocking: %d\n", errno);
                exit(1);
        }
    } /* while */
}

/*   add-books program*/

/*
 * 1. Read a new book entry from screen.
 * 2. Insert book in book file.
 * 3. Use semaphore "wrtsem" to block new readers.
 * 4. Wait for semaphore "rdsem" to reach 0.
 * 5. Insert book into index.
 * 6. Decrement wrtsem.
 * 7. Go to 1.
 */
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include "liber.h"

void exit();
extern int errno;
struct sembuf sop[1];
BOOK bookbuf;

main()
{
    .
    .
    .
    for (;;)
    {

        /*
         * Read information on new book from screen.
         */
```

```
        addscr(&bookbuf);

        /* write new record at the end of the bookfile.
         * Code not shown, but
         * addscr() returns a 1 if title information has
         * been entered, 0 if not.
         */

        /*
         * Increment write semaphore, blocking new readers from
         * accessing the index.
         */
        sop[0].sem_flg = 0;
        sop[0].sem_op = 1;
        if (semop(wrtsem, sop, 1) == -1)
        {
                (void) fprintf(stderr, "semop failed: %d\n", errno);
                exit(1);
        }
        /*
         * Wait for read semaphore to reach 0 (all readers to finish
         * using the index).
         */
        sop[0].sem_op = 0;
        if (semop(rdsem, sop, 1) == -1)
        {
                (void) fprintf(stderr, "semop failed: %d\n", errno);
                exit(1);
        }
        /*
         * Now that we have exclusive access to the index we
         * insert our new book with its file pointer.
         */

        /* Decrement write semaphore, permitting readers to read index. */
        sop[0].sem_op = -1;
        if (semop(wrtsem, sop, 1) == -1)
        {
                (void) fprintf(stderr, "semop failed: %d\n", errno);
                exit(1);
        }
    } /* for */
```

```
    .
    .
    .
}
```

The example following, **addscr()**, illustrates two significant points about **curses** screens:

1. Information read in from a **curses** window can be stored in fields that are part of a structure defined in the header file for the application.
2. The address of the structure can be passed from another function where the record is processed.

```
                /*  addscr is called from add-books.
                 *  The user is prompted for title
                 *  information.
                 */
#include <curses.h>

WINDOW *cmdwin;

addscr(bb)
struct BOOK *bb;
{
    int c;

    initscr();
    nonl();
    noecho();
    cbreak();

    cmdwin = newwin(6, 40, 3, 20);
    mvprintw(0, 0, "This screen is for adding titles to the database");
    mvprintw(1, 0, "Enter  a  to add;  q  to quit: ");
    refresh();
    for (;;)
    {

        refresh();
        c = getch();
        switch (c) {
```

```
            case 'a':
                    werase(cmdwin);
                    box(cmdwin, '|', '-');
                    mvwprintw(cmdwin, 1, 1, "Enter title: ");
                    wmove(cmdwin, 2, 1);
                    echo();
                    wrefresh(cmdwin);
                    wgetstr(cmdwin, bb->title);
                    noecho();
                    werase(cmdwin);
                    box(cmdwin, '|', '-');
                    mvwprintw(cmdwin, 1, 1, "Enter author: ");
                    wmove(cmdwin, 2, 1);
                    echo();
                    wrefresh(cmdwin);
                    wgetstr(cmdwin, bb->author);
                    noecho();
                    werase(cmdwin);
                    wrefresh(cmdwin);
                    endwin();
                    return(1);
            case 'q':
                    erase();
                    endwin();
                    return(0);
            }
    }
}

#
# Makefile for liber library system
#

CC = cc
CFLAGS = -O
all: startup add-books checkout card-catalog

startup: liber.h startup.c
    $(CC) $(CFLAGS) -o startup startup.c

add-books: add-books.o addscr.o
    $(CC) $(CFLAGS) -o add-books add-books.o addscr.o
```

```
add-books.o: liber.h

checkout: liber.h checkout.c
    $(CC) $(CFLAGS) -o checkout checkout.c

card-catalog: liber.h card-catalog.c
    $(CC) $(CFLAGS) -o card-catalog card-catalog.c
```

Chapter 3
File and Record Locking

3.1 Introduction 3-1

3.2 Terminology 3-1

3.3 File Protection 3-3
Opening a File for Record Locking 3-3
Setting a File Lock 3-4
Setting and Removing Record Locks 3-7
Getting Lock Information 3-11
Deadlock Handling 3-13

3.4 Selecting Advisory or Mandatory Locking 3-14
Things to Remember About Mandatory Locking 3-15

Chapter 3
File and Record Locking

3.1 Introduction

This chapter describes how record locking can be used, what it can do, and what it cannot do. This facility meets and exceeds the */usr/group* record locking specifications in all its capabilities. The */usr/group* interface is implemented as a library function and is totally compatible with all */usr/group* features.

Examples are given for the correct use of record locking. Misconceptions about the amount of protection that record locking affords are dispelled. Record locking should be viewed as a synchronization mechanism, not a security mechanism.

The manual pages for the **fcntl**(2) system call, the **lockf**(3) library function, and **fcntl**(5) data structures and commands are referred to throughout this chapter. You should read them before continuing.

3.2 Terminology

Before discussing how record locking should be used, let us first define a few terms.

Record	A contiguous set of bytes in a file. The operating system does not impose any record structure on files. This may be done by the programs that use the files.
Cooperating Processes	Processes that work together in some well-defined fashion to accomplish the tasks at hand. Processes that share files must request permission to access the files before using them. File access permissions must be carefully set to restrict non-cooperating processes from accessing those files. The term *process* will be used interchangeably with *cooperating process* to refer to a task obeying such protocols.

Read (Share) Locks

These are used to gain limited access to sections of files. When a read lock is in place on a record, other processes may also read lock that record, in whole or in part. No other process, however, may have or obtain a write lock on an overlapping section of the file. If a process holds a read lock, it may assume that no other process will be writing or updating that record at the same time. This access method also permits many processes to read the given record. This might be necessary when searching a file, without the contention involved if a write or exclusive lock were to be used.

Write (Exclusive) Locks

These are used to gain complete control over sections of files. When a write lock is in place on a record, no other process may read or write lock that record, in whole or in part. If a process holds a write lock, it may assume that no other process will be reading or writing that record at the same time.

Advisory Locking

A form of record locking that does not interact with the I/O subsystem (for example, **creat**(2), **open**(2), **read**(2), and **write**(2)). The control over records is accomplished by requiring an appropriate record lock request before I/O operations. If appropriate requests are always made by all processes accessing the file, then the accessibility of the file will be controlled by the interaction of these requests. Advisory locking depends on the individual processes to enforce the record locking protocol; it does not require an accessibility check at the time of each I/O request.

Mandatory Locking

A form of record locking that does interact with the I/O subsystem. Access to locked records is enforced by the **creat**(2), **open**(2), **read**(2), and **write**(2) system calls. If a record is locked, then access of that record by any other process is restricted according to the type of lock on the record. The control over records should still be performed explicitly by requesting an appropriate record lock before I/O operations, but an additional check is made by the system before each I/O

operation to ensure the record locking protocol is being honored. Mandatory locking offers an extra synchronization check, but at the cost of some additional system overhead.

3.3 File Protection

There are access permissions for system files to control who may read, write, or execute such a file. These access permissions may only be set by the owner of the file or by the superuser. The permissions of the directory in which the file resides can also affect the ultimate disposition of a file. Note that if the directory permissions allow anyone to write in it, then files within the directory may be removed, even if those files do not have read, write, or execute permission for that user. If your application warrants the use of record locking, make sure that the permissions on your files and directories are set properly. A record lock, even a mandatory record lock, will only protect the portions of the files that are locked. Other parts of these files might be corrupted if proper precautions are not taken.

Only a known set of programs and administrators should be able to read or write a database. This can be done easily by setting the set-group-ID bit (see **chmod**(1)) of the database accessing programs. The files can then be accessed by a known set of programs that obey the record locking protocol. An example of such file protection, although record locking is not used, is the **mail**(1) command. In that command only the particular user and the **mail** command can read and write in the unread mail files.

3.3.1 Opening a File for Record Locking

The first requirement for locking a file or segment of a file is having a valid open file descriptor. If read locks are to be done, then the file must be opened with at least read accessibility and likewise for write locks and write accessibility. Below is an example of opening our file for both read and write access:

```
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>


int fd;       /* file descriptor */
char *filename;
```

```
main(argc, argv)
int argc;
char *argv[];
{
    extern void exit(), perror();

    /* get database file name from command line and open the
     * file for read and write access.
     */
    if (argc < 2) {
        (void) fprintf(stderr, "usage: %s filename\n", argv[0]);
        exit(2);
    }
    filename = argv[1];
    fd = open(filename, O_RDWR);
    if (fd < 0) {
        perror(filename);
        exit(2);
    }
        .
        .
        .
        .
```

The file is now open for us to perform both locking and I/O functions. We can then proceed with the task of setting a lock.

3.3.2 Setting a File Lock

There are several ways for us to set a lock on a file. In part, these methods depend upon how the lock interacts with the rest of the program. There are also questions of performance as well as portability. Two methods will be given here, one using the **fcntl**(2) system call, the other using the */usr/group* standards compatible **lockf**(3) library function call.

Locking an entire file is just a special case of record locking. For both these methods the concept and the effect of the lock are the same. The file is locked starting at a byte offset of zero (0) until the end of the maximum file size. This point extends beyond any real end of the file so that no lock can be placed on this file beyond this point. To do this, the value of the size of the lock is set to zero. The next example shows code using the **fcntl**(2) system call.

```
#include <fcntl.h>
#define MAX_TRY  10
int try;
struct flock lck;

try = 0;

/* set up the record locking structure, the address of which
 * is passed to the fcntl system call.
 */
lck.l_type = F_WRLCK;    /* setting a write lock */
lck.l_whence = 0; /* offset l_start from beginning of file */
lck.l_start = 0L;
lck.l_len = 0L;       /* until the end of the file address space */

/* Attempt locking MAX_TRY times before giving up.
 */
while (fcntl(fd, F_SETLK, &lck) < 0) {
    if (errno == EAGAIN || errno == EACCES) {
        /* there might be other errors cases in which
         * you might try again.
         */
        if (++try < MAX_TRY) {
            (void) sleep(2);
            continue;
        }
        (void) fprintf(stderr,"File busy try again later!\n");
        return;
    }
    perror("fcntl");
    exit(2);
}
    .
    .
    .
```

This portion of code tries to lock a file. This is attempted several times until one of the following things happens:

- The file is locked.

- An error occurs.
- It gives up trying because MAX_TRY has been exceeded.

To perform the same task using the **lockf**(3) function, see the following example.

```
#include <unistd.h>
#define MAX_TRY  10
int try;
try = 0;

/* make sure the file pointer
 * is at the beginning of the file.
 */
lseek(fd, 0L, 0);

/* Attempt locking MAX_TRY times before giving up.
 */
while (lockf(fd, F_TLOCK, 0L) < 0) {
    if (errno == EAGAIN || errno == EACCES) {
        /* there might be other errors cases in which
         * you might try again.
         */
        if (++try < MAX_TRY) {
            sleep(2);
            continue;
        }
        (void) fprintf(stderr,"File busy try again later!\n");
        return;
    }
    perror("lockf");
    exit(2);
}
    .
    .
```

It should be noted that the **lockf**(3) example appears to be simpler, but the **fcntl**(2) example exhibits additional flexibility. Using the **fcntl**(2) method, it is possible to set the type and start of the lock request simply by setting a few structure variables. **lockf**(3) merely sets write (exclusive) locks; an additional system call (**lseek**(2)) is required to specify the start of the lock.

3.3.3 Setting and Removing Record Locks

Locking a record is done the same way as locking a file except for the differing starting point and length of the lock. We will now try to solve an interesting and real problem. There are two records (these records may be in the same or different file) that must be updated simultaneously so that other processes get a consistent view of this information. (This type of problem comes up, for example, when updating the interrecord pointers in a doubly linked list.) To do this you must decide the following questions:

- What do you want to lock?
- For multiple locks, in what order do you want to lock and unlock the records?
- What do you do if you succeed in getting all the required locks?
- What do you do if you fail to get all the locks?

In managing record locks, you must plan a failure strategy if one cannot obtain all the required locks. It is because of contention for these records that we have decided to use record locking in the first place. Different programs might respond in the following ways:

- Wait a certain amount of time, and try again
- Abort the procedure and warn the user
- Let the process sleep until signaled that the lock has been freed
- Some combination of the above

Let us now look at our example of inserting an entry into a doubly linked list. For the example, we will assume that the record after which the new record is to be inserted has a read lock on it already. The lock on this record must be changed or promoted to a write lock so that the record may be edited.

Promoting a lock (generally from read lock to write lock) is permitted if no other process is holding a read lock in the same section of the file. If there are processes with pending write locks that are sleeping on the same section of the file, the lock promotion succeeds and the other (sleeping) locks wait. Promoting (or demoting) a write lock to a read lock carries no restrictions. In either case, the lock is merely reset with the new lock type. Because the */usr/group* **lockf** function does not have read locks, lock promotion is not applicable to that call. The next example shows an example of record locking with lock promotion.

```
struct record {
    .
    .       /* data portion of record */
    .
    long prev;   /* index to previous record in the list */
    long next;   /* index to next record in the list */
};
/* Lock promotion using fcntl(2)
 * When this routine is entered it is assumed that there are read
 * locks on "here" and "next".
 * If write locks on "here" and "next" are obtained:
 *    Set a write lock on "this".
 *    Return index to "this" record.
 * If any write lock is not obtained:
 *    Restore read locks on "here" and "next".
 *    Remove all other locks.
 *    Return a -1.
 */
long
set3lock (this, here, next)
long this, here, next;
{
    struct flock lck;

    lck.l_type = F_WRLCK;     /* setting a write lock */
    lck.l_whence = 0;     /* offset l_start from beginning of file */
    lck.l_start = here;
    lck.l_len = sizeof(struct record);

    /* promote lock on "here" to write lock */
    if (fcntl(fd, F_SETLKW, &lck) < 0) {
        return (-1);
    }
    /* lock "this" with write lock */
    lck.l_start = this;
    if (fcntl(fd, F_SETLKW, &lck) < 0) {
        /* Lock on "this" failed;
         * demote lock on "here" to read lock.
         */
        lck.l_type = F_RDLCK;
        lck.l_start = here;
        (void) fcntl(fd, F_SETLKW, &lck);
```

```
        return (-1);
    }
    /* promote lock on "next" to write lock */
    lck.l_start = next;
    if (fcntl(fd, F_SETLKW, &lck) < 0) {
        /* Lock on "next" failed;
         * demote lock on "here" to read lock,
         */
        lck.l_type = F_RDLCK;
        lck.l_start = here;
        (void) fcntl(fd, F_SETLK, &lck);
        /* and remove lock on "this".
         */
        lck.l_type = F_UNLCK;
        lck.l_start = this;
        (void) fcntl(fd, F_SETLK, &lck);
        return (-1); /* cannot set lock, try again or quit */
    }

    return (this);
}
```

The locks on these three records were all set to wait (sleep) if another process was blocking them from being set. This was done with the F_SETLKW command. If the F_SETLK command was used instead, the **fcntl** system calls would fail if blocked. The program would then have to be changed to handle the blocked condition in each of the error return sections.

Let us now look at a similar example using the **lockf** function. Since there are no read locks, all (write) locks will be referenced generically as locks.

```
/* Lock promotion using lockf(3)
 * When this routine is entered it is assumed that
 * there are no locks on "here" and "next".
 * If locks are obtained:
 *     Set a lock on "this".
 *     Return index to "this" record.
 * If any lock is not obtained:
 *     Remove all other locks.
 *     Return a -1.
 */
```

```
#include <unistd.h>

long
set3lock (this, here, next)
long this, here, next;

{

    /* lock "here" */
    (void) lseek(fd, here, 0);
    if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
        return (-1);
    }
    /* lock "this" */
    (void) lseek(fd, this, 0);
    if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
        /* Lock on "this" failed.
         * Clear lock on "here".
         */
        (void) lseek(fd, here, 0);
        (void) lockf(fd, F_ULOCK, sizeof(struct record));
        return (-1);

    }

    /* lock "next" */
    (void) lseek(fd, next, 0);
    if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {

        /* Lock on "next" failed.
         * Clear lock on "here",
         */
        (void) lseek(fd, here, 0);
        (void) lockf(fd, F_ULOCK, sizeof(struct record));

        /* and remove lock on "this".
         */
        (void) lseek(fd, this, 0);
        (void) lockf(fd, F_ULOCK, sizeof(struct record));
        return (-1);  /* cannot set lock, try again or quit */

    }
```

```
    return (this);
}
```

Locks are removed in the same manner as they are set, only the lock type is different (F_UNLCK or F_ULOCK). An unlock cannot be blocked by another process and will only affect locks that were placed by this process. The unlock only affects the section of the file defined in the previous example by **lck**. It is possible to unlock or change the type of lock on a subsection of a previously set lock. This may cause an additional lock (two locks for one system call) to be used by the operating system. This occurs if the subsection is from the middle of the previously set lock.

3.3.4 Getting Lock Information

One can determine which processes, if any, are blocking a lock from being set. This can be used as a simple test or as a means to find locks on a file. A lock is set up as in the previous examples and the F_GETLK command is used in the **fcntl** call. If the lock passed to **fcntl** would be blocked, the first blocking lock is returned to the process through the structure passed to **fcntl**. That is, the lock data passed to **fcntl** is overwritten by blocking lock information. This information includes two pieces of data that have not been discussed yet, **l_pid** and **l_sysid**, that are only used by F_GETLK. (For systems that do not support a distributed architecture, the value in **l_sysid** should be ignored.) These fields uniquely identify the process holding the lock.

If a lock passed to **fcntl** using the F_GETLK command would not be blocked by another process's lock, then the **l_type** field is changed to F_UNLCK and the remaining fields in the structure are unaffected. We'll use this capability to print all the segments locked by other processes. Note in the next example that if there are several read locks over the same segment, only one of these will be found.

```
struct flock lck;

/*Find and print "write lock" blocked segments of this file.*/
    (void) printf("sysid   pid type    start   length\n");
    lck.l_whence = 0;
    lck.l_start = 0L;
    lck.l_len = 0L;
    do {
        lck.l_type = F_WRLCK;
        (void) fcntl(fd, F_GETLK, &lck);
        if (lck.l_type != F_UNLCK) {
```

```
            (void) printf("%5d %5d    %c   %8d %8d\n",
                lck.l_sysid,
                lck.l_pid,
                (lck.l_type == F_WRLCK) ? 'W' : 'R',
                lck.l_start,
                lck.l_len);
            /* if this lock goes to the end of the address
             * space, no need to look further, so break out.
             */
            if (lck.l_len == 0)
                break;
            /* otherwise, look for new lock after the one
             * just found.
             */
            lck.l_start += lck.l_len;
        }
    } while (lck.l_type != F_UNLCK);
```

fcntl with the F_GETLK command will always return correctly (that is, it will not sleep or fail) if the values passed to it as arguments are valid.

The **lockf** function with the F_TEST command can also be used to test if there is a process blocking a lock. This function does not, however, return the information about where the lock actually is and which process owns the lock. The next example shows a routine using **lockf** to test for a lock on a file.

```
/* find a blocked record. */

/* seek to beginning of file */
(void) lseek(fd, 0, 0L);
/* set the size of the test region to zero (0)
 * to test until the end of the file address space.
 */
if (lockf(fd, F_TEST, 0L) < 0) {
    switch (errno) {
    case EACCES:
    case EAGAIN:
        (void) printf("file is locked by another process\n");
        break;
    case EBADF:
        /* bad argument passed to lockf */
        perror("lockf");
```

```
            break;
        default:
            (void) printf("lockf: unknown error <%d>\n", errno);
            break;
        }
}
```

When a process forks, the child receives a copy of the file descriptors that the parent has opened. The parent and child also share a common file pointer for each file. If the parent were to seek to a point in the file, the child's file pointer would also be at that location. This feature has important implications when using record locking. The current value of the file pointer is used as the reference for the offset of the beginning of the lock, as described by **l_start**, when using a **l_whence** value of 1. If both the parent and child process set locks on the same file, there is a possibility that a lock will be set using a file pointer that was reset by the other process. This problem appears in the **lockf**(3) function call as well and is a result of the */usr/group* requirements for record locking. If forking is used in a record locking program, the child process should close and reopen the file if either locking method is used. This will result in the creation of a new and separate file pointer that can be manipulated without this problem occurring. Another solution is to use the **fcntl** system call with a **l_whence** value of 0 or 2. This makes the locking function atomic, so that even processes sharing file pointers can be locked without difficulty.

3.3.5 Deadlock Handling

There is a certain level of deadlock detection/avoidance built into the record locking facility. This deadlock handling provides the same level of protection granted by the */usr/group* standard **lockf** call. This deadlock detection is only valid for processes that are locking files or records on a single system. Deadlocks can only occur when the system is about to put a record locking system call to sleep. A search is made for constraint loops of processes that would cause the system call to sleep indefinitely. If such a situation is found, the locking system call will fail and set **errno** to the deadlock error number. If a process wishes to avoid the use of the systems deadlock detection, it should set its locks using F_GETLK instead of F_GETLKW.

3.4 Selecting Advisory or Mandatory Locking

The use of mandatory locking is not recommended for reasons that will be made clear in a subsequent section. Whether or not locks are enforced by the I/O system calls is determined at the time the calls are made and the state of the permissions on the file (see **chmod**(2)). For locks to be under mandatory enforcement, the file must be a regular file with the set-group-ID bit on and the group execute permission off. If either condition fails, all record locks are advisory. Mandatory enforcement can be assured by the code shown in the next example.

```
#include <sys/types.h>
#include <sys/stat.h>

int mode;
struct stat buf;
        .
        .
        .
    if (stat(filename, &buf) < 0) {
        perror("program");
        exit (2);
    }
    /* get currently set mode */
    mode = buf.st_mode;
    /* remove group execute permission from mode */
    mode &= ~(S_IEXEC>>3);
    /* set 'set group id bit' in mode */
    mode |= S_ISGID;
    if (chmod(filename, mode) < 0) {
        perror("program");
        exit(2);
    }
        .
        .
        .
```

Files that are to be record-locked should never have any type of execute permission set on them. This is because the operating system does not obey the record locking protocol when executing a file.

The **chmod**(1) command can also be easily used to set a file to have mandatory locking. This can be done with the command:

chmod +l *filename*

The **ls**(1) command was also changed to show this setting when you ask for the long listing format:

ls -l *filename*

causes the following to be printed:

```
-rw---l---   1 abc        other     1048576 Feb  30 11:44 filename
```

3.4.1 Things to Remember About Mandatory Locking

- Mandatory locking only protects those portions of a file that are locked. Other portions of the file that are not locked may be accessed according to normal system file permissions.
- If multiple reads or writes are necessary for an atomic transaction, the process should explicitly lock all such pieces before any I/O begins. Thus advisory enforcement is sufficient for all programs that perform in this way.
- As stated earlier, arbitrary programs should not have unrestricted access permission to files that are important enough to record lock.
- Advisory locking is more efficient because a record lock check does not have to be performed for every I/O request.

Chapter 4
Interprocess Communication

4.1 Introduction 4-1
Keys and IDs 4-1

4.2 Messages 4-4
Getting Message Queues 4-8
Using **msgget** 4-8
Controlling Message Queues 4-14
Using **msgctl** 4-14
Sample Program 4-15
Operations for Messages 4-19
Using **msgop** 4-20
Sample Program 4-22

4.3 Semaphores 4-29
Using Semaphores 4-31
Getting Semaphores 4-35
Using **semget** 4-35
Sample Program 4-37
Controlling Semaphores 4-41
Using **semctl** 4-41
Sample Program 4-43
Operations on Semaphores 4-52
Using **semop** 4-52
Sample Program 4-53

4.4 Shared Memory 4-57
Using Shared Memory 4-58
Getting Shared Memory Segments 4-62
Using **shmget** 4-62
Sample Program 4-65
Controlling Shared Memory 4-68
Using **shmctl** 4-68
Sample Program 4-69
Operations for Shared Memory 4-74
Using **shmop** 4-74
Sample Program 4-76

Tables

4-1 Message Operation Permission Codes 4-9
4-2 Control Commands (Flags) 4-10
4-3 Operation Permissions Codes 4-36
4-4 **semget** Control Commands (Flags) 4-36
4-5 Shared Memory State Information 4-60
4-6 shmflg Operation Permissions Codes 4-63
4-7 shmget Control Commands (Flags) 4-64

Chapter 4
Interprocess Communication

4.1 Introduction

The operating system supports three types of Interprocess Communication (IPC):

- Messages
- Semaphores
- Shared memory

This chapter describes the system calls for each type of IPC. Included are several example programs that show the use of the IPC system calls. Since the C Programming Language provides many ways to accomplish the same task or requirement, keep in mind that the sample programs were written for clarity and not for program efficiency.

4.1.1 Keys and IDs

The system calls described in this chapter all require either a **key** argument, or an **id** argument. This section discusses the **key** argument and the associated IDs.

The operating system IPC primitives present to the user two types of "names," or ways to identify a specific IPC object (a message queue, a semaphore list, or a shared memory segment). These names are referred to as "keys" and "ids."

A **key** is an integer value passed as the first argument to **semget()**, **msgget()**, or **shmget()**. system calls. Keys are analogous to file names, in that they specify a "name" to an object. When an application needs to communicate through a common IPC object, it must choose, in advance, a **key** to identify that object. Developers must be careful to choose **key** values that don't conflict with choices made by other programmers doing unrelated applications.

An application passes a **key** to a **semget()**, **msgget()**, or **shmget()** system call, along with a **flags** argument specifying operation permissions for the object and options regarding IPC object creation (IPC_CREAT, IPC_EXCL). The key IPC_PRIVATE (value zero) is special in that it always generates a new IPC object.

Upon successful completion (after checking operation permissions, for example) the ***get()** system calls return a positive integer called an **id.** **Id**s are analogous to file descriptors in that they are generated by the kernel and returned via a system call. The system call does a lookup operation using the name you specified, and returns a value that is used for all subsequent operations on that object.

However, **id**s are different from file descriptors in that they can be passed to other processes and used to operate on the IPC object. File descriptors are valid only within the context of the process for which they were created and that process's descendants.

All IPC operations (**shmat()**, **shmdt()**, **shmctl()**, **semop()**, **semctl()**, **msgsnd()**, **msgrcv()**, and **msgctl()**) on the IPC object use an **id** to identify the object. The value of an **id** associated with a **key** is determined by the kernel when that object is created. Once an IPC object has been created, all ***get()** operations using the **key** for that object will return the same **id** value.

The **id** value is valid only as long as that particular IPC object exists. If that object is deleted (IPC_RMID), and re-created some time later using the same **key** value, the **id** returned from ***get** will be *not* be the same as before.

A model of usage might be something like this:

program1 does a ***get()** system call to create an IPC object with KEYVAL as the **key** *name*. The kernel creates the IPC object and assigns it an **id** with value IDVAL.

program2 and *program3* want to communicate with each other using the IPC object identified by KEYVAL. Each program does a ***get()** system call, passing KEYVAL as the first argument. Since the object already exists, the ***get()** system call will return IDVAL to *program1* and *program2*. These programs now use IDVAL to communicate with each other by doing operations on the object.

If *program3* now deletes the object identified by KEYVAL, the **id** value IDVAL is invalidated. Running *program1* again to create an IPC object with **key** value KEYVAL will actually create a *new* IPC object, with a *new* **id** value. *program2* and *program3*

can learn this new **id** value by doing another ***get()** system call with argument KEYVAL.

The **key** value IPC_PRIVATE (value of zero) is a special value. It *always* creates a new IPC object (with a new **id** value). This can be useful for applications that don't want to be concerned with preagreeing on **key** values (or colliding on key values with other applications). The process that creates the object (**id = *get(IPC_PRIVATE,...)**) can then communicate (for example, by forking) the **id** value returned to all the processes that want to communicate using that object. As before, this **id** value is valid only as long as that IPC object exists. When it is deleted, that **id** value will be invalidated.

Observe that another process cannot learn the **id** of an object with **key** IPC_PRIVATE by doing another *get() with the IPC_PRIVATE key. It would succeeds only in creating yet another new object with a different **id**.

The IPC creation flags mentioned above (IPC_CREAT, IPC_EXCL) work in the following manner (for key values other that IPC_PRIVATE):

If neither flag is set, the ***get(***keyval***,...)** system call will test to see if an object with **key** *keyval* already exists. If it does, and the operation permissions passed to the system call pass permission checks, the system call will succeed, and the **id** value returned will be that associated with the object found. If there is no IPC object with **key** *keyval*, the system call will fail, returning value (-1).

If only the IPC_CREAT flag is set, the ***get** system call will test to see if an object with *keyval* already exists. If it does, the system call will behave just as in the case outlined above (now with *flags* set). If the object does *not* exist, the system call will create one, initializing it with information from other arguments to the system call, and will return a new **id** value.

If both the IPC_CREAT flag and IPC_EXCL flags are set, the system call will test to see if an object with *keyval* already exists. If it does, the system call will fail, returning value (-1). If the object does not exist, the system call will create a new object, initialize it with other information from the creating process and system call arguments, and return a new **id** value for that object.

If the IPC_CREAT flag is *not* set, the IPC_EXCL flag is ignored.

4.2 Messages

Discrete packets of data transmitted between processes (executing programs) are called messages. Processes using the message type of IPC can perform two operations:

- Sending
- Receiving

Before processes can send or receive messages, the process must have the operating system generate the necessary software mechanisms. A process does this with the **msgget**(2) system call. While doing this, the process becomes the owner/creator of the message facility and specifies the initial operation permissions for that message queue. Subsequently, the owner/creator can relinquish ownership or change the operation permissions using the **msgctl**(2) system call. However, the creator remains the creator as long as that message queue exists. Other processes, with proper permission, can use **msgctl**() to perform various other control functions.

Processes that are attempting to send or receive a message can suspend execution if their operations are unsuccessful. That is, a sending process can wait until the receiving process is ready to communicate and vice versa. A process specifying a suspension of execution is a "blocking message operation." A process not allowing a suspension of execution is a "nonblocking message operation."

A blocking message operation can be suspended until one of three conditions occurs:

- It is successful.
- It receives a signal.
- The facility is removed.

System calls make these message capabilities available to processes. The calling process passes arguments to a system call, and the system call either successfully or unsuccessfully performs its function. If the system call is successful, it performs its function and returns applicable information. Otherwise, a known error code (–1) is returned to the process, and an external error number variable **errno** is set accordingly.

Before a message can be sent or received, a message queue and an associated data structure must be created. Each message queue has assigned to it a unique message queue identifier (**msqid**).

The message queue stores (header) information about each message being sent or received. For each message, this information includes the following:

- Pointer to the next message on queue
- Message type
- Message text size
- Message text address

The data structure associated with the message queue contains the following information:

- Operation permissions data (operation permission structure)
- Pointer to first message on the queue
- Pointer to last message on the queue
- Current number of bytes on the queue
- Number of messages on the queue
- Maximum number of bytes on the queue
- Process identification (PID) of last message sender
- PID of last message receiver
- Last message send time
- Last message receive time
- Last change time

NOTE

All include files discussed in this chapter are located in the /usr/include *or* /usr/include/sys *directories.*

The data structure definition for the per-message header information contained in the message queue is as follows:

```
struct msg
{
  struct msg *msg_next;  /* ptr to next message on q */
  long    msg_type;    /* message type */
  short       msg_ts;      /* message text size */
  short       msg_spot;    /* message text map address */
};
```

This data structure definition is located in the */usr/include/sys/msg.h* header file.

The data structure definition for the message queue itself is ass follows:

```
struct msqid_ds
{
  struct ipc_perm  msg_perm;    /* operation permission struct */
  struct msg       *msg_first;  /* ptr to first message on q */
  struct msg       *msg_last;   /* ptr to last message on q */
  ushort           msg_cbytes;  /* current # bytes on q */
  ushort           msg_qnum;    /* # of messages on q */
  ushort           msg_qbytes;  /* max # of bytes on q */
  ushort           msg_lspid;   /* pid of last msgsnd */
  ushort           msg_lrpid;   /* pid of last msgrcv */
  time_t           msg_stime;   /* last msgsnd time */
  time_t           msg_rtime;   /* last msgrcv time */
  time_t           msg_ctime;   /* last change time */
};
```

The message queue data structure is also located in the **#include <sys/msg.h>** header file. Note that the **msg_perm** member of this structure uses **ipc_perm** as a template. The breakout for the operation permissions data structure is shown in the definition of the **ipc_perm** data structure:

```
struct ipc_perm
{
    ushort uid;   /* owner's user id */
    ushort gid;   /* owner's group id */
    ushort cuid;  /* creator's user id */
    ushort cgid;  /* creator's group id */
    ushort mode;  /* access modes */
    ushort seq;   /* slot usage sequence number */
```

```
    key_t  key;    /* key */
};
```

The **ipc_perm** data structure is located in the **#include <sys/ipc.h>** header file and is common for all IPC facilities.

The **msgget**(2) system call performs one of two tasks when only the IPC_CREAT flag is set in the **msgflg** argument it receives:

- Get a new **msqid** and create an associated message queue and data structure for it, and return the **msqid**.
- Return an existing **msqid** that already has an associated message queue and data structure

The value of the **key** argument passed to the **msgget**() system call determines the task performed. A new **msqid** is returned with an associated message queue and data structure if the value for **key** is not already being used by an existing **msqid**, and if no system tunable parameters are exceeded (for example, limit on number of message queues in system).

There is also a provision for specifying a **key** value of zero. This is known as the private **key** (IPC_PRIVATE = 0). When a private **key** is specified, a new **msqid** is always returned with an associated message queue and data structure unless a system tunable parameter would be exceeded. For security reasons, when the **ipcs** command is performed, the KEY field in the **msqid** is all zeros.

If a **msqid** already exists for the **key** specified, the value of the existing **msqid** is returned. If you do not want an existing **msqid** returned, a control command (IPC_EXCL) can be specified (set) in the **msgflg** argument passed to the system call. The details of using this system call are discussed in "Using **msgget**," later in this chapter.

When getting a new **msqid** and creating an associated message queue and data structure for it, the process calling **msgget** becomes the owner/creator, and the associated data structure is initialized accordingly. Remember, ownership can be changed but the creating process always remains the creator (see "Controlling Message Queues" later in this chapter). The creator of the message queue also determines the initial operation permissions for it.

Once a message queue and data structure are created, the message operations [**msgop**()] and message control [**msgctl**()] system calls can be used on that queue using that queue's unique identifier (**msqid**).

Message operations, as mentioned previously, consist of sending and receiving messages. These operations are invoked with **msgsnd**() and **msgrcv**() system calls. Refer to "Operations for Messages" later in this chapter for details.

Message control is done by using the **msgctl**(2) system call. It permits you to control the message facility in the following ways:

- Change operation permissions for a message queue.
- Change the **size** (**msg_qbytes**) of the message queue for a particular **msqid**.
- Remove a particular **msqid** from the operating system along with its associated message queue and data structure.

Refer to "Controlling Message Queues" later in this chapter for details of the **msgctl**() system call.

4.2.1 Getting Message Queues

This section explains how to use the **msgget**(2) system call and provides a sample program illustrating its use.

Using msgget

The synopsis found in the **msgget**(2) man page is as follows:

```
#include  <sys/types.h>
#include  <sys/ipc.h>
#include  <sys/msg.h>

int  msgget (key, msgflg)
key_t  key;
int msgflg;
```

Upon successful completion, the integer returned from this function is the message queue identifier (**msqid**) that was discussed earlier. As declared, the process calling the **msgget**() system call must supply two arguments to be passed as the **key** and **msgflg** arguments.

A new **msqid** with an associated message queue and data structure is returned if either of these conditions is met:

- **key** is equal to IPC_PRIVATE.
- **key** is a unique integer, and **msgflg** ANDed with IPC_CREAT is TRUE.

The value passed to the **msgflg** argument must be an integer and specify the following items:

- Access permissions
- Execution modes
- Control fields (commands)

The **msgflg** argument contains a specification of the read/write (send/receive) access permissions for the owner/group/other attributes. These attributes are collectively referred to as "operation permissions." Table 4-1 reflects the numeric values (expressed in octal notation) for the valid operation permissions codes.

Table 4-1
Message Operation Permission Codes

Operation Permissions	Octal Value
Read by Owner	00400
Write by Owner	00200
Read by Group	00040
Write by Group	00020
Read by Others	00004
Write by Others	00002

A specific value is derived by adding the values for the operation permissions desired. That is, if read by owner and read/write by others is desired, the code value would be 00406 (00400 plus 00004 plus 00002). There are constants located in the *msg.h* header file for the owner.

The *msgflgs* argument also contains control command flags. These flags are given names to make using them easier. Table 4-2 contains the names and values of the constants that apply to the **msgget**() system call. They may also be referred to as flags and are defined in the *ipc.h* header file.

Table 4-2
Control Commands (Flags)

Control Command	Value
IPC_CREAT	0001000
IPC_EXCL	0002000

The value for **msgflg** is, therefore, a combination of operation permissions and control commands (flags). The desired flag(s) can be specified after determining the value for the operation permissions. This is accomplished by bit-wise ORing (|) the control command flags with the operation permissions.

The **msgflg** value can be easily set by using the names of the flags in conjunction with the octal operation permissions value:

```
msqid = msgget (key, (IPC_CREAT | 0400));

msqid = msgget (key, (IPC_CREAT | IPC_EXCL | 0400));
```

As specified by the **msgget**(2) man page, success or failure of this system call depends upon the argument values for **key** and **msgflg**. Success is also dependent on the system tunable parameters. The system call attempts to return a new **msqid** if one of the following conditions is true:

- Condition 1 — **key** is equal to IPC_PRIVATE (0)
- Condition 2 — **key** does not already have a **msqid** associated with it, and (**msgflg** & IPC_CREAT) is *true* (not zero).

The **key** argument can be set to IPC_PRIVATE in the following ways:

```
msqid = msgget (IPC_PRIVATE, msgflg);

msqid = msgget ( 0 , msgflg);
```

This alone initiates the system call because it satisfies condition 1. The system tunable parameter MSGMNI determines the maximum number of unique message queues (**msqids**) in the operating system. Exceeding the value set for MSGMNI always causes a failure.

Condition 2 is satisfied if the value for **key** is not already associated with a **msqid** and the bit-wise ANDing of **msgflg** and IPC_CREAT is true (not zero). This means that, for this facility type, the **key** is unique (not in use) within the operating system and that the IPC_CREAT flag is set (**msgflg** | IPC_CREAT).

IPC_EXCL is another control command used in conjunction with IPC_CREAT. When IPC_EXCL is set, the system call will fail if, and only if, a **msqid** exists for the specified **key**. When both IPC_CREAT and IPC_EXCL are specified, a new **msqid** is returned if the system call is successful. Refer to the **msgget**(2) man page for the specific associated data structure initialization. The specific failure conditions with error names are contained there also.

Sample Program

The sample program in this section is a menu-driven program that exercises all possible combinations of the **msgget**(2) system call.

Lines 4-8 of the program show the required header files as specified by the **msgget**(2) man page. The variables declared for this program are as follows:

key	The value for the desired **key**.
opperm	Stores the desired operation permissions.
flags	Stores the desired control commands (flags).
opperm_flags	Stores the combination from the logical OR of the **opperm** and **flags** variables; it is then used in the system call to pass the **msgflg** argument.
msqid	The message queue identification number for a successful system call or the error code (–1) for an unsuccessful one.

```
1  /*This is a program to illustrate
2  **the message get, msgget(),
3  **system call capabilities.*/

4   #include    <stdio.h>
5   #include    <sys/types.h>
6   #include    <sys/ipc.h>
7   #include    <sys/msg.h>
8   #include    <errno.h>
```

```
9   /*Start of main C language program*/
10  main()
11  {
12      key_t key;              /*declare as long integer*/
13      int opperm, flags;
14      int msqid, opperm_flags;
15   /*Enter the desired key*/
16   printf("Enter the desired key in hex = ");
17    scanf("%x", &key);


18      /*Enter the desired octal operation
19        permissions.*/
20        printf("\nEnter the operation\n");
21        printf("permissions in octal = ");
22        scanf("%o", &opperm);
23      /*Set the desired flags.*/
24      printf("\nEnter corresponding number to\n");
25      printf("set the desired flags:\n");
26      printf("No flags                    = 0\n");
27      printf("IPC_CREAT                   = 1\n");
28      printf("IPC_EXCL                    = 2\n");
29      printf("IPC_CREAT and IPC_EXCL      = 3\n");
30      printf("            Flags           = ");

31      /*Get the flag(s) to be set.*/
32      scanf("%d", &flags);

33      /*Check the values.*/
34      printf ("\nkey =0x%x, opperm = 0%o, flags = 0%o\n",
35          key, opperm, flags);

36      /*Incorporate the control fields (flags) with
37        the operation permissions*/
38      switch (flags)
39      {
40      case 0:     /*No flags are to be set.*/
41        opperm_flags = (opperm | 0);
42        break;
43      case 1:     /*Set the IPC_CREAT flag.*/
44          opperm_flags = (opperm | IPC_CREAT);
45          break;
```

```
46      case 2:     /*Set the IPC_EXCL flag.*/
47          opperm_flags = (opperm | IPC_EXCL);
48          break;
49      case 3: /*Set the IPC_CREAT and IPC_EXCL flags.*/
50        opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
51    }
52    /*Call the msgget system call.*/
53    msqid = msgget (key, opperm_flags);

54    /*Perform if the call is unsuccessful.*/
55    if(msqid == -1)
56      {
57          printf ("\nThe msgget system call failed!\n");
58          printf ("The error number = %d\n", errno);
59      }

60      /*Return the msqid upon successful completion.*/
61      else
62          printf ("\nThe msqid = %d\n", msqid);
63      exit(0);
64  }
```

The sample program begins by prompting for a hexadecimal **key**, an octal operation permissions code, and finally for the control command combinations (flags) which are selected from a menu (lines 15-32). All possible combinations are allowed even though they might not be viable. This allows you to observe the errors resulting from illegal combinations.

Next, the menu selection for the flags is combined with the operation permissions, and the result is stored in the **opperm_flags** variable (lines 36-51).

The **msgget()** system call is made next, and the result is stored in the **msqid** variable (line 53).

Since the **msqid** variable now contains a valid message queue identifier or the error code (–1), it is tested to see if an error occurred (line 55). If **msqid** equals –1, a message indicates that an error resulted, and the value of the external **errno** variable is displayed (lines 57, 58).

If no error occurred, the returned message queue identifier is displayed (line 62).

4.2.2 Controlling Message Queues

This section explains how to use the **msgctl** system call and provides a sample program that exercises all of its capabilities.

4.2.3 Using msgctl

The synopsis found in the **msgctl**(2) man page is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl (msqid, cmd, buf)
int msqid, cmd;
struct msqid_ds *buf;
```

The **msgctl**() system call requires three arguments, and it returns an integer value. Upon successful completion, a zero value is returned; when unsuccessful, it returns a –1.

msqid Must be a nonnegative integer that was returned from a **msgget**() system call.

cmd Can be replaced by one of the following control commands (flags):

- IPC_STAT Return the status information contained in the associated data structure for the specified **msqid**, and place it in the data structure pointed to by the ***buf** pointer in the user memory area.
- IPC_SET For the specified **msqid**, set the effective owner and group identification, operation permissions, and the number of bytes for the message queue.
- IPC_RMID Remove the specified **msqid** along with its associated message queue and data structure.

A process must have an effective user identification equal to the owner or creator uids for the message queue, or be owned by a superuser in order to

perform an IPC_SET or IPC_RMID control command. Read permission is required to perform the IPC_STAT control command.

The details of this system call are discussed in the sample program. If you have problems understanding the logic in this program, read "Using **msgget**" later in this chapter.

Sample Program

The example program in this section is a menu-driven program that exercises all possible combinations of the **msgctl**(2) system call.

Lines 5-9 show the required header files as specified by the **msgctl**(2) man page. The variables declared for this program are as follows:

uid	Stores the value for the effective user identification.
gid	Stores the value for the effective group identification.
mode	Stores the value for the operation permissions.
bytes	Stores the value for the number of bytes in the message queue (**msg_qbytes**).
	The variables *uid*, *gid*, *mode*, and *bytes* demonstrate the IPC_SET command in the example program.
rtrn	Stores the return integer value from the system call.
msqid	Stores the message queue identifier for the **msgclt()** system call.
command	Stores the code for the desired control command so that subsequent processing can be performed on it.
choice	Determines which message queue structure member is changed for the IPC_SET control command.
msqid_ds	Receives a copy of the specified message queue identifier's data structure when an IPC_STAT control command is performed.
***buf**	A pointer to a **msqid_ds** structure.

The sample program for the **msgctl**() system call follows:

```
1  /*This is a program to illustrate
2  **the message control, msgctl(),
```

```
3  **system call capabilities.
4  */

5  /*Include necessary header files.*/
6  #include    <stdio.h>
7  #include    <sys/types.h>
8  #include    <sys/ipc.h>
9  #include    <sys/msg.h>

10  /*Start of main C language program*/
11  main()
12  {
13      extern int errno;
14      int uid, gid, mode, bytes;
15      int rtrn, msqid, command, choice;
16      struct msqid_ds msqid_ds, *buf;
17      buf = &msqid_ds;

18      /*Get the msqid, and command.*/
19      printf("Enter the msqid = ");
20      scanf("%d", &msqid);
21      printf("\nEnter the number for\n");
22      printf("the desired command:\n");
23      printf("IPC_STAT    =  1\n");
24      printf("IPC_SET     =  2\n");
25      printf("IPC_RMID    =  3\n");
26      printf("Entry       =  ");
27      scanf("%d", &command);
28      /*Check the values.*/
29      printf ("\nmsqid =%d, command = %d\n",
30          msqid, command);

31      switch (command)
32      {
33      case 1:     /*Use msgctl() to duplicate
34            the data structure for
35                    msqid in the msqid_ds area pointed
36                    to by buf and then print it out.*/
37          rtrn = msgctl(msqid, IPC_STAT,
38              buf);
39          printf ("\nThe USER ID = %d\n",
40              buf->msg_perm.uid);
```

```
41          printf ("The GROUP ID = %d\n",
42              buf->msg_perm.gid);
43          printf ("The operation permissions = 0%o\n",
44              buf->msg_perm.mode);
45          printf ("The msg_qbytes = %d\n",
46              buf->msg_qbytes);
47          break;
48      case 2:     /*Select and change the desired
49                     member(s) of the data structure.*/
50          /*Get the original data for this msqid
51                data structure first.*/
52          rtrn = msgctl(msqid, IPC_STAT, buf);
53          printf("\nEnter the number for the\n");
54          printf("member to be changed:\n");
55          printf("msg_perm.uid    = 1\n");
56          printf("msg_perm.gid    = 2\n");
57          printf("msg_perm.mode   = 3\n");
58          printf("msg_qbytes      = 4\n");
59          printf("Entry           = ");
60          scanf("%d", &choice);
61          /*Only one choice is allowed per
62            pass as an illegal entry will
63                cause repetitive failures until
64            msqid_ds is updated with
65                IPC_STAT.*/

66          switch(choice){
67          case 1:
68              printf("\nEnter USER ID = ");
69              scanf ("%d", &uid);
70              buf->msg_perm.uid = uid;
71              printf("\nUSER ID = %d\n",
72                  buf->msg_perm.uid);
73              break;
74          case 2:
75              printf("\nEnter GROUP ID = ");
76              scanf("%d", &gid);
77              buf->msg_perm.gid = gid;
78              printf("\nGROUP ID = %d\n",
79                  buf->msg_perm.gid);
80              break;
81          case 3:
```

```
82              printf("\nEnter MODE = ");
83              scanf("%o", &mode);
84              buf->msg_perm.mode = mode;
85              printf("\nMODE = 0%o\n",
86                  buf->msg_perm.mode);
87              break;
88          case 4:
89              printf("\nEnter msq_bytes = ");
90              scanf("%d", &bytes);
91              buf->msg_qbytes = bytes;
92              printf("\nmsg_qbytes = %d\n",
93                  buf->msg_qbytes);
94              break;
95            }

96          /*Do the change.*/
97          rtrn = msgctl(msqid, IPC_SET,
98              buf);
99          break;
100     case 3:     /*Remove the msqid along with its
101                    associated message queue
102                    and data structure.*/
103         rtrn = msgctl(msqid, IPC_RMID, NULL);
104     }
105     /*Perform the following if call unsuccessful.*/
106     if(rtrn == -1)
107     {
108         printf ("\nThe msgctl system call failed!\n");
109         printf ("The error number = %d\n", errno);
110     }
111     /*Return the msqid upon successful completion.*/
112     else
113         printf ("\nMsgctl successful for msqid = %d\n",
114             msqid);
115     exit (0);
116 }
```

First, the program prompts for a valid message queue identifier and stores that value in the **msqid** variable (lines 19, 20). This is required for every **msgctl** system call.

Then the code for the desired control command must be entered (lines 21-27), and it is stored into the command variable. The code is tested to determine the control command for subsequent processing.

If the IPC_STAT control command is selected (code 1), the **msgcnt()** system call is performed (lines 37, 38) and the status information returned is printed out (lines 39-46); only the members that can be set are printed out in this program. Note that if the system call is unsuccessful (line 106), the status information of the last successful call is printed. In addition, an error message is displayed and the **errno** variable is printed out (lines 108, 109). If the system call is successful, a message indicates this, along with the message queue identifier (lines 111-114).

If the IPC_SET control command is selected (code 2), the first thing done is to get the current status information for the message queue identifier specified (lines 50-52). This is necessary because this sample program provides for changing only one member at a time, and the system requires values for all the fields. Also, if an invalid value happened to be stored in the user memory area for one of these members, it would cause repetitive failures for this control command until corrected. The next thing the program does is to prompt for a code corresponding to the member to be changed (lines 53-59). This code is stored in the **choice** variable (line 60). Now, depending upon the member picked, the program prompts for the new value (lines 66-95). The value is placed into the appropriate member in the user memory area data structure, and the system call is made (lines 96-98). Depending upon success or failure, the program returns the same messages as for IPC_STAT above.

If the IPC_RMID control command (code 3) is selected, the system call is performed (lines 100-103), and the **msqid** (along with its associated message queue and data structure) is removed from the operating system. Note that the ***buf** pointer is not required as an argument to perform this control command. Depending upon the success or failure, the program returns the same messages as for the other control commands.

4.2.4 Operations for Messages

This section explains how to use the **msgsnd**(2) and **msgrcv**(2) system calls, and provides a sample program that exercises all of their capabilities.

Using msgop

The synopsis found in the **msgop**(2) man page is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd (msqid, msgp, msgsz, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz, msgflg;

int msgrcv (msqid, msgp, msgsz, msgtyp, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz;
long msgtyp;
int msgflg;
```

Sending a Message. The **msgsnd** system call requires that four arguments be passed to it. Upon successful completion, a zero value is returned. When unsuccessful, **msgsnd**() returns a –1.

The value for the **msqid** argument must be a nonnegative integer returned from a **msgget**() system call.

The **msgp** argument is a pointer to a structure in the user memory area that contains the type of the message and the message to be sent.

The **msgsz** argument specifies the length of the character array in the data structure pointed to by the **msgp** argument. This is the length of the message. The maximum **size** of this array is determined by the system tunable parameter MSGMAX .

The **msg_qbytes** data structure member can be lowered from MSGMNB by using the **msgctl**() IPC_SET control command, but only the superuser can raise it afterwards.

The **msgflg** argument allows the "blocking message operation" to be performed if the IPC_NOWAIT flag is not set (**msgflg** & IPC_NOWAIT = 0). This occurs if the total number of bytes allowed on the specified message queue are in use (**msg_qbytes** or MSGMNB), or the total system-wide number of messages on all queues is equal to the system imposed limit (MSGTQL). If the IPC_NOWAIT flag is set, the system call will fail and return a –1.

Further details of the **msgsnd()** system call are shown in the next sample program.

Receiving Messages. The **msgrcv()** system call requires that five arguments to be passed it: **msqid, msgp, msgsz, msgtyp,** and **msgflg.** Upon successful completion, a value equal to the number of bytes received is returned and when unsuccessful it returns a –1. The five arguments are as follows:

msqid — Must be a nonnegative integer value returned from the **msgget()** system call.

msgp — A pointer to a structure in the user memory area that will receive the message type and the message text.

msgsz — Specifies the length of the message to be received. If its value is less than the message in the array, an error can be returned if desired (see the **msgflg** argument).

msgtyp — Picks the first message on the message queue of the particular type specified. If it is equal to zero, the first message on the queue is received; if it is greater than zero, the first message of the same type is received; if it is less than zero, the lowest type that is less than or equal to its absolute value is received.

msgflg — Allows the "blocking message operation" to be performed if the IPC_NOWAIT flag is not set (**msgflg** & IPC_NOWAIT = 0). This occurs if there is no message to be received on the message queue of the desired type (**msgtyp**). If the IPC_NOWAIT flag is set, the system call fails immediately when there is no message of the desired type on the queue. **msgflg** can also make sure the system call fails if the message is longer than the **size** to be received. This is done by not setting the MSG_NOERROR flag in the **msgflg** argument (**msgflg** & MSG_NOERROR = 0). If the MSG_NOERROR flag is set, the message is truncated to the length specified by the **msgsz** argument of **msgrcv()**.

Further details of the **msgrcv()** system call are shown in the next sample program.

Sample Program

The sample program in this section is a menu-driven program that exercises combinations of the **msgsnd**() and **msgrcv**(2) system calls.

Lines 5-9 show the required header files as specified by the **msgop**(2) man page. The variables declared for this program are:

sndbuf A buffer containing the message to be sent (line 13); it uses the **msgbuf1** data structure as a template (lines 10-13). The **msgbuf1** structure (lines 10-13) is almost an exact duplicate of the **msgbuf** structure contained in the **msg.h** header file. The only difference is that the character array for **msgbuf1** contains the maximum message **size** (MSGMAX) for the computer where in **msgbuf** it is set to one (1) to satisfy the compiler. For this reason **msgbuf** cannot be used directly as a template for the user-written program. It is there so you can determine its members.

rcvbuf A buffer to receive the message (line 13); it uses the **msgbuf1** data structure as a template (lines 10-13).

***msgp** A pointer (line 13) to both the **sndbuf** and **rcvbuf** buffers.

i A counter for inputting characters from the keyboard, storing them in the array, and keeping track of the message length for the **msgsnd**() system call; it is also used as a counter to output the received message for the **msgrcv**() system call.

c Receives the input character from the **getchar**() function (line 50).

flag Stores the code of IPC_NOWAIT for the **msgsnd**() system call (line 61).

flags Stores the code of the IPC_NOWAIT or MSG_NOERROR flags for the **msgrcv**() system call (line 117).

choice Stores the code for sending or receiving (line 30).

rtrn Stores the return values from all system calls.

msqid Stores and passes the desired message queue identifier for both system calls.

msgsz Stores and passes the **size** of the message to be sent or received.

msgflg Passes the value of flag for sending or the value of flags for receiving.

msgtyp Specifies the message type for sending, or picks a message type for receiving.

```
1  /*This is a program to illustrate
2  **the message operations, msgop(),
3  **system call capabilities.
4  */

5  /*Include necessary header files.*/
6  #include    <stdio.h>
7  #include    <sys/types.h>
8  #include    <sys/ipc.h>
9  #include    <sys/msg.h>

10  struct msgbuf1 {
11      long    mtype;
12      char    mtext[8192];
13  } sndbuf, rcvbuf, *msgp;

14  /*Start of main C language program*/
15  main()
16  {
17      extern int errno;
18      int i, c, flag, flags, choice;
19      int rtrn, msqid, msgsz, msgflg;
20      long mtype, msgtyp;
21      struct msqid_ds msqid_ds, *buf;
22      buf = &msqid_ds;
23      /*Select the desired operation.*/
24      printf("Enter the corresponding\n");
25      printf("code to send or\n");
26      printf("receive a message:\n");
27      printf("Send              =  1\n");
28      printf("Receive           =  2\n");
29      printf("Entry             =  ");
30      scanf("%d", &choice);

31      if(choice == 1) /*Send a message.*/
32      {
```

```
33          msgp = &sndbuf; /*Point to user send structure.*/

34          printf("\nEnter the msqid of\n");
35          printf("the message queue to\n");
36          printf("handle the message = ");
37          scanf("%d", &msqid);

38          /*Set the message type.*/
39          printf("\nEnter a positive integer\n");
40          printf("message type (long) for the\n");
41          printf("message = ");
42          scanf("%d", &msgtyp);
43          msgp->mtype = msgtyp;

44          /*Enter the message to send.*/
45          printf("\nEnter a message: \n");

46          /*A control-d (^d) terminates as
47            EOF.*/
48          /*Get each character of the message
49            and put it in the mtext array.*/
50          for(i = 0; ((c = getchar()) != EOF); i++)
51              sndbuf.mtext[i] = c;

52          /*Determine the message size.*/
53          msgsz = i + 1;

54          /*Echo the message to send.*/
55          for(i = 0; i < msgsz; i++)
56              putchar(sndbuf.mtext[i]);

57          /*Set the IPC_NOWAIT flag if
58            desired.*/
59          printf("\nEnter a 1 if you want the\n");
60          printf("the IPC_NOWAIT flag set:  ");
61          scanf("%d", &flag);
62          if(flag == 1)
63              msgflg |= IPC_NOWAIT;
64          else
65              msgflg = 0;

66          /*Check the msgflg.*/
```

```
67          printf("\nmsgflg = 0%o\n", msgflg);

68        /*Send the message.*/
69          rtrn = msgsnd(msqid, msgp, msgsz, msgflg);
70          if(rtrn == -1)
71          printf("\nMsgsnd failed.  Error = %d\n",
72                  errno);
73          else {
74              /*Print the value of test which
75                    should be zero for successful.*/
76              printf("\nValue returned = %d\n", rtrn);
77              /*Print the size of the message
78                sent.*/
79              printf("\nMsgsz = %d\n", msgsz);

80              /*Check the data structure update.*/
81              msgctl(msqid, IPC_STAT, buf);

82                /*Print out the affected members.*/

83              /*Print the incremented number of
84                messages on the queue.*/
85              printf("\nThe msg_qnum = %d\n",
86                  buf->msg_qnum);
87              /*Print the process id of the last sender.*/
88              printf("The msg_lspid = %d\n",
89                  buf->msg_lspid);
90              /*Print the last send time.*/
91              printf("The msg_stime = %d\n",
92                  buf->msg_stime);
93          }
94      }

95      if(choice == 2)  /*Receive a message.*/
96      {
97          /*Initialize the message pointer
98            to the receive buffer.*/
99          msgp = &rcvbuf;

100          /*Specify the message queue which contains
101                the desired message.*/
102          printf("\nEnter the msqid = ");
```

```
103         scanf("%d", &msqid);
104         /*Specify the specific message on the queue
105               by using its type.*/
106         printf("\nEnter the msgtyp = ");
107         scanf("%d", &msgtyp);

108         /*Configure the control flags for the
109               desired actions.*/
110         printf("\nEnter the corresponding code\n");
111         printf("to select the desired flags: \n");
112         printf("No flags                      =  0\n");
113         printf("MSG_NOERROR                   =  1\n");
114         printf("IPC_NOWAIT                    =  2\n");
115         printf("MSG_NOERROR and IPC_NOWAIT    =  3\n");
116         printf("                Flags         =  ");
117         scanf("%d", &flags);

118         switch(flags) {
119             /*Set msgflg by ORing it with the
120                   appropriate flags (constants).*/
121         case 0:
122             msgflg = 0;
123             break;
124         case 1:
125             msgflg |= MSG_NOERROR;
126             break;
127         case 2:
128             msgflg |= IPC_NOWAIT;
129             break;
130         case 3:
131             msgflg |= MSG_NOERROR | IPC_NOWAIT;
132             break;
133         }
134         /*Specify the number of bytes to receive.*/
135         printf("\nEnter the number of bytes\n");
136         printf("to receive (msgsz) = ");
137         scanf("%d", &msgsz);

138         /*Check the values for the arguments.*/
139         printf("\nmsqid =%d\n", msqid);
140         printf("\nmsgtyp = %d\n", msgtyp);
141         printf("\nmsgsz = %d\n", msgsz);
```

```
142             printf("\nmsgflg = 0%o\n", msgflg);

143             /*Call msgrcv to receive the message.*/
144             rtrn = msgrcv(msqid, msgp, msgsz, msgtyp, msgflg);

145             if(rtrn == -1)  {
146                 printf("\nMsgrcv failed.  ");
147                 printf("Error = %d\n", errno);
148             }
149             else {
150                 printf ("\nMsgctl was successful\n");
151                 printf("for msqid = %d\n",
152                     msqid);

153                 /*Print the number of bytes received,
154                   it is equal to the return
155                   value.*/
156                 printf("Bytes received = %d\n", rtrn);
157                 /*Print the received message.*/
158                 for(i = 0; i<=rtrn; i++)
159                     putchar(rcvbuf.mtext[i]);
160             }
161             /*Check the associated data structure.*/
162             msgctl(msqid, IPC_STAT, buf);
163             /*Print the decremented number of messages.*/
164             printf("\nThe msg_qnum = %d\n", buf->msg_qnum);
165             /*Print the process id of the last receiver.*/
166             printf("The msg_lrpid = %d\n", buf->msg_lrpid);
167             /*Print the last message receive time*/
168             printf("The msg_rtime = %d\n", buf->msg_rtime);
169         }
170     }
```

Note that a **msqid_ds** data structure is set up in the program (line 21) with a pointer which is initialized to point to it (line 22). Structure members are observed by using the **msgctl**() (IPC_STAT) system call to get them for the program to print them out (lines 80-92 and lines 161-168).

The first thing the program prompts for is whether to send or receive a message. A corresponding code must be entered for the desired operation, and it is stored in the **choice** variable (lines 23-30). Depending upon the code, the program proceeds as in the following **msgsnd** or **msgrcv** sections.

msgsnd. When the code is to send a message, the **msgp** pointer is initialized (line 33) to the address of the send data structure, **sndbuf**. Next, a message type must be entered for the message; it is stored in the variable **msgtyp** (line 42), and then (line 43) it is put into the **mtype** member of the data structure pointed to by **msgp**.

The program now prompts for a message to be entered from the keyboard and enters a loop of getting and storing into the **mtext** array of the data structure (lines 48-51). This will continue until an end-of-file is recognized, which for the **getchar**() function is a control-d (CTRL-D) immediately following a carriage return (<CR>). When this happens, the **size** of the message is determined by adding 1 to the **i** counter (lines 52, 53) as it stored the message beginning in the zero array element of **mtext**. Keep in mind that the message also contains the terminating characters, and the message will therefore appear to be three characters short of **msgsz**.

The message is immediately echoed from the **mtext** array of the **sndbuf** data structure to provide feedback (lines 54-56).

The final thing that must be decided is whether to set the IPC_NOWAIT flag. The program does this by requesting that a code of 1 be entered for yes and anything else for no (lines 57-65). It is stored in the **flag** variable. If a 1 is entered, IPC_NOWAIT is logically ORed with **msgflg**; otherwise, **msgflg** is set to zero.

The **msgsnd**() system call is performed (line 69). If it is unsuccessful, a failure message is displayed along with the error number (lines 70-72). If it is successful, the returned value is printed; this value should be zero (lines 73-76).

Every time a message is successfully sent, there are three members of the message queue data structure that are updated. They are as follows:

msg_qnum	Represents the total number of messages on the message queue; it is incremented by one.
msg_lspid	Contains the Process Identification (PID) number of the last process sending a message; it is set accordingly.
msg_stime	Contains the time in seconds since January 1, 1970, Greenwich Mean Time (GMT) of the last message sent; it is set accordingly.

These members are displayed after every successful message send operation (lines 79-92).

msgrcv. If the code specifies that a message is to be received, the program continues execution as described in the following paragraphs.

The **msgp** pointer is initialized to the **rcvbuf** data structure (line 99).

Next, the message queue identifier of the sending message queue is requested, and it is stored in **msqid** (lines 100-103).

The message type is requested; it is stored in **msgtyp** (lines 104-107).

The code for the desired combination of control flags is requested next; it is stored in **nflags** (lines 108-117). Depending upon the selected combination, **msgflg** is set accordingly (lines 118-133).

Finally, the number of bytes to be received is requested; it is stored in **msgsz** (lines 134-137).

The **msgrcv**() system call is performed (line 144). If it is unsuccessful, a message and error number is displayed (lines 145-148). If successful, a message indicates so, and the number of bytes returned is displayed, followed by the received message (lines 153-159).

When a message is successfully received, there are three members of the associated data structure that are updated. They are as follows:

msg_qnum	Contains the number of messages on the message queue; it is decremented by one.
msg_lrpid	Contains the process identification (PID) of the last process receiving a message; it is set accordingly.
msg_rtime	Contains the time in seconds since January 1, 1970, Greenwich Mean Time (GMT) that the last process received a message; it is set accordingly.

4.3 Semaphores

The semaphore type of Interprocess Communication (IPC) allows processes to communicate through the exchange of semaphore values. A semaphore is a positive integer (0 through 32,767). Since many applications require the use of more than one semaphore, the operating system has the ability to create sets or arrays of semaphores. A semaphore set can contain one or more semaphores up to a limit set by the system administrator. The tunable parameter SEMMSL has a default value of 25. Semaphore sets are created by using the **semget**(2) system call.

The process performing the **semget**(2) system call becomes the owner/creator, determines how many semaphores are in the set, and sets the operation permissions for the set. This process can subsequently relinquish ownership of the set or change the operation permissions using the **semctl**(), semaphore control, system call. The creating process always remains the creator as long as that set exists. Other processes with proper permission can use **semctl**() to perform other control functions.

Provided a process has alter permission, it can manipulate the semaphore(s) in that set. Each semaphore within a set can be manipulated in two ways with the **semop**(2) system call:

- Increment the semaphore value
- Decrement the semaphore value

To increment a semaphore, an integer value of the desired magnitude is passed to the **semop**(2) system call. To decrement a semaphore, a negative (–) value of the desired magnitude is passed.

The operating system ensures that only one process can manipulate a semaphore set at any given time. Simultaneous requests are performed sequentially in an arbitrary order.

A process atomically tests and modifies the value of a semaphore. The process requests that the semaphore be decremented by a value *n*. If the semaphore's value is greater than or equal to the decrement value, the request succeeds and the semaphore value is decremented. Otherwise, the semaphore value is not modified. While doing this, the process can have its execution suspended (IPC_NOWAIT flag not set) until the semaphore value permits the operation (other processes increment the semaphore) or the semaphore facility is removed.

Suspending execution is called a "blocking semaphore operation." This capability is also available for a process that is testing for a semaphore to become zero or equal to zero. Only read permission is required for this test; it is accomplished by passing a value of zero to the **semop**(2) system call.

On the other hand, if the process request is not successful and the process does not request to have its execution suspended, it is called a "nonblocking semaphore operation." In this case, a known error code (–1) is returned to the the process and the external **errno** variable is set accordingly.

The blocking semaphore operation allows interprocess communication on the basis of semaphore values at different points in time. Remember also that IPC objects remain in the operating system until removed by a permitted process or until the system is reinitialized.

When a set of semaphores is created, the first semaphore in the set is semaphore number 0. The last semaphore number in the set is one less than the total number of semaphores in the set. For example, in a set of five semaphores, the first would be 0 and the last 4.

An array of these "blocking/nonblocking operations" can be performed on a set containing more than one semaphore. When performing an array of operations, the "blocking/nonblocking operations" can be applied to any or all of the semaphores in the set. Also, the operations can be applied in any order of semaphore number. However, no operations are done until all operations in the request can be done successfully. This requirement means that preceding changes made to semaphore values in the set must be undone when an operation on a semaphore in the set cannot be completed successfully. For example, if a process has successfully completed three of six operations on a set of ten semaphores but cannot perform the fourth operation, no changes are made to the set until the fourth and remaining operations are successfully performed.

Remember, any unsuccessful "nonblocking operation" for a single semaphore or a set of semaphores returns immediately, with no operations performed at all. When this occurs, a known error code (–1) returns to the process and the external variable **errno** is set accordingly.

4.3.1 Using Semaphores

Before semaphores can be used (operated on or controlled), a uniquely identified *data structure* and *semaphore set* (array) must be created. The unique identifier is called the semaphore identifier (**semid**) and is used to identify or reference a particular data structure and semaphore set.

The semaphore set contains a predefined number of structures in an array, one structure for each semaphore in the set. The number of semaphores (**nsems**) in a semaphore set is selectable. The following members are in each structure within a semaphore set:

- Semaphore value

- Process identification (PID) performing last operation
- Number of processes awaiting the semaphore value to become greater than its current value
- Number of processes awaiting the semaphore value to equal zero

There is one associated data structure for the uniquely identified semaphore set. This data structure contains information related to the semaphore set:

- Operation permissions data (operation permissions structure)
- Pointer to first semaphore in the set (array)
- Number of semaphores in the set
- Last semaphore operation time
- Last semaphore change time

The data structure definition for the semaphore set (array member) is as follows:

```
struct sem
{
   ushort   semval;          /* semaphore value */
   short    sempid;          /* pid of last operation */
   ushort   semncnt;         /* # awaiting semval > cval */
   ushort   semzcnt;         /* # awaiting semval = 0 */
};
```

This structure definition is located in the **#include <sys/sem.h>** header file.

Likewise, the structure definition for the associated semaphore data structure is as follows:

```
struct semid_ds
{
   struct ipc_perm sem_perm;   /* operation permission struct */
   struct sem      *sem_base;  /* ptr to first semaphore in set */
   ushort          sem_nsems;  /* # of semaphores in set */
   time_t          sem_otime;  /* last semop time */
   time_t          sem_ctime;  /* last change time */
};
```

This structure definition is also located in the **#include <sys/sem.h>** header file. Note that the **sem_perm** member of this structure uses **ipc_perm** as a template. The breakout for the operation permissions data structure is shown in **ipc_perm** data structure shown in the "Messages" section, earlier in ths chapter. The **ipc_perm** data structure is the same for all IPC facilities, and it is located in the **#include <sys/ipc.h>** header file.

The **semget**(2) system call performs one of two tasks when only the IPC_CREAT flag is set in the **semflg** argument it receives:

- Task 1 — Get a new **semid** and create an associated data structure and semaphore set for it.
- Task 2 — Return an existing **semid** that already has an associated data structure and semaphore set.

The value of the **key** argument passed to the **semget**(2) system call determines the task performed. For task 1, if the **key** is not already in use for an existing **semid**, a new **semid** is returned with an associated data structure and semaphore set, provided no tunable system parameter is exceeded.

There is also a provision for specifying a **key** of value zero (0). This is known as the private **key** (IPC_PRIVATE = 0). When specified, a new **semid** is always returned with an associated data structure and semaphore set unless a system tunable parameter is exceeded. When the **ipcs** command is performed, the KEY field for the **semid** is all zeros.

When performing task 1, the process calling **semget**() becomes the owner/creator, and the associated data structure is initialized accordingly. Remember, ownership can be changed, but the creating process always remains the creator (see "Controlling Semaphores" later in this chapter). The creator of the semaphore set also determines the initial operation permissions for the facility.

For task 2, if a **semid** exists for the **key** specified, the value of the existing **semid** is returned. If you do not want an existing **semid** returned, a control command (IPC_EXCL) can be specified (set) in the **semflg** argument passed to the system call. The system call fails if it is passed a value for the number of semaphores (**nsems**) that is greater than the number actually in the set. If you do not know how many semaphores are in the set, use 0 for **nsems**. The details of using this system call are discussed in the "Using **semget**" section later in this chapter.

Once a uniquely identified semaphore set and data structure are created, semaphore operations [**semop**(2)] and semaphore control [**semctl**()] can be performed using the **semid** returned form **semget**().

Semaphore operations consist of incrementing, decrementing, and testing for zero. A single system call, **semop**(), performs these operations. Refer to "Operations on Semaphores" later in this chapter for details of this system call.

Semaphore control is preformed with the **semctl**(2) system call. These control operations permit you to control the semaphore facility in the following ways:

- Return the value of a semaphore.
- Set the value of a semaphore.
- Return the process identification (PID) of the last process performing an operation on a semaphore set.
- Return the number of processes waiting for a semaphore value to become greater than its current value.
- Return the number of processes waiting for a semaphore value to equal zero.
- Get all semaphore values in a set and place them in an array in user memory.
- Set all semaphore values in a semaphore set from an array of values in user memory.
- Place all data structure member values and status of a semaphore set into user memory area.
- Change operation permissions for a semaphore set.
- Remove a particular **semid** from the operating system along with its associated data structure and semaphore set.

Refer to "Controlling Semaphores" later in this chapter for details of the **semctl**(2) system call.

4.3.2 Getting Semaphores

This section contains a detailed description of using the **semget**(2) system call along with a sample program illustrating its use.

Using semget

The synopsis found in the **semget**(2) man page is as follows:

```
#include  <sys/types.h>
#include  <sys/ipc.h>
#include  <sys/sem.h>

int  semget (key, nsems, semg)
key_t  key;
int nsems, semg;
```

Upon successful completion, the integer returned from this system call is the semaphore set identifier (**semid**) discussed earlier.

A new **semid** with an associated semaphore set and data structure is provided if either of these two conditions are met:

- Condition 1 — **key** is equal to IPC_PRIVATE.
- Condition 2 — **key** is passed a unique integer, and **semflg** ANDed with IPC_CREAT is TRUE.

The value passed as the **semflg** argument must specify the following:

- Access permissions
- Control fields (commands)

Access permissions determine the read/alter operations permitted for the owner/group/other classes of processes. They are collectively referred to as "operation permissions." Table 4-3 reflects the numeric values (expressed in octal notation) for the valid operation permissions codes.

Table 4-3
Operation Permissions Codes

Operation Permissions	Octal Value
Read by Owner	00400
Alter by Owner	00200
Read by Group	00040
Alter by Group	00020
Read by Others	00004
Alter by Others	00002

Operation permissions are derived by adding the values of each operation permission desired. That is, if read by user and read/alter by others is desired, the code value would be 00406 (00400 plus 00004 plus 00002). There are constants defined in the **sem.h** header file that can be used for the user (OWNER). They are as follows:

```
SEM_A     0200     /* alter permission by owner */
SEM_R     0400     /* read permission by owner */
```

Control commands are predefined constants (represented by all uppercase letters). Table 4-4 contains the names and values of the constants that apply to the **semget**(2) system call. They are also referred to as flags and are defined in the **ipc.h** header file.

Table 4-4
semget Control Commands (Flags)

Control Command	Value
IPC_CREAT	0001000
IPC_EXCL	0002000

The value for **semflg** is, therefore, a combination of operation permissions and control commands. The desired flag(s) can be specified after determining the operation permissions value. This is accomplished by a bitwise OR (|) of the flag values with the operation permissions.

As specified by the **semget**(2) man page, success or failure of this system call depends upon the actual argument values for **key**, **nsems**, **semflg**, or the pertinent system tunable parameters. The system call will attempt to return a new **semid** if one of the following conditions is true:

- Condition 1 — **key** is equal to IPC_PRIVATE (0).
- Condition 2 — **key** does not already have a **semid** associated with it, and (**semflg** & IPC_CREAT). is *true* (not zero).

The **key** argument can be set to IPC_PRIVATE in the following way:

```
semid = semget (IPC_PRIVATE, nsems, semflg);
```

Exceeding the SEMMNI, SEMMNS, or SEMMSL system tunable parameters will always cause a failure. The SEMMNI system tunable parameter determines the maximum number of unique semaphore sets (**semid**) in the operating system. The SEMMNS system tunable parameter determines the maximum number of semaphores in all semaphore sets system-wide. The SEMMSL system tunable parameter determines the maximum number of semaphores in each semaphore set.

Condition 2 is satisfied if the value for **key** is not already associated with a **semid**, and the bitwise ANDing of **semflg** and IPC_CREAT is true (not zero). This means that, for this facility type, **key** is unique (not in use) within the operating system and that the IPC_CREAT flag is set (**semflg** | IPC_CREAT).

IPC_EXCL is another control command used in conjunction with IPC_CREAT. If IPC_EXCL is set, the **semget**() system call fails if, and only if, a **semid** exists for the specific **key** provided. This is necessary to prevent the process from thinking that it has received a new (unique) **semid** when it has not. When both IPC_CREAT and IPC_EXCL are specified, a new **semid** is returned if the system call is successful. Any value for **semflg** returns a new **semid** if **key** equals zero (IPC_PRIVATE) and no system tunable parameters are exceeded.

Refer to the **semget**(2) manual page for more information.

Sample Program

The example program in this section is a menu-driven program that exercises all possible combinations of the **semget**(2) system call.

Lines 4-8 in the sample program include the required header files as specified in the **semget**(2) man page.

The variables declared for this program are as follows:

key	The value for the desired key.
opperm	Stores the desired operation permissions.
flags	Stores the desired control commands (flags).
opperm_flags	Stores the combination from the logical ORing of the **opperm** and **flags** variables; then passes the **semflg** argument in the system call.
semid	The semaphore set identification number for a successful system call or the error code (–1) for an unsuccessful one.

The sample program for the **semget**(2) system call is shown here:

```
1  /*This is a program to illustrate
2  **the semaphore get, semget(),
3  **system call capabilities.*/

4  #include    <stdio.h>
5  #include    <sys/types.h>
6  #include    <sys/ipc.h>
7  #include    <sys/sem.h>
8  #include    <errno.h>

9  /*Start of main C language program*/
10  main()
11  {
12       key_t key;      /*declare as long integer*/
13       int opperm, flags, nsems;
14       int semid, opperm_flags;

15       /*Enter the desired key*/
16       printf("\nEnter the desired key in hex = ");
17       scanf("%x", &key);

18       /*Enter the desired octal operation
19            permissions.*/
20       printf("\nEnter the operation\n");
```

```
21      printf("permissions in octal = ");
22      scanf("%o", &opperm);
23      /*Set the desired flags.*/
24      printf("\nEnter corresponding number to\n");
25      printf("set the desired flags:\n");
26      printf("No flags                    = 0\n");
27      printf("IPC_CREAT                   = 1\n");
28      printf("IPC_EXCL                    = 2\n");
29      printf("IPC_CREAT and IPC_EXCL      = 3\n");
30      printf("            Flags           = ");
31      /*Get the flags to be set.*/
32      scanf("%d", &flags);

33      /*Error checking (debugging)*/
34      printf ("\nkey =0x%x, opperm = 0%o, flags = 0%o\n",
35          key, opperm, flags);
36      /*Incorporate the control fields (flags) with
37            the operation permissions.*/
38      switch (flags)
39      {
40      case 0:     /*No flags are to be set.*/
41          opperm_flags = (opperm | 0);
42          break;
43      case 1:     /*Set the IPC_CREAT flag.*/
44          opperm_flags = (opperm | IPC_CREAT);
45          break;
46      case 2:     /*Set the IPC_EXCL flag.*/
47          opperm_flags = (opperm | IPC_EXCL);
48          break;
49      case 3: /*Set the IPC_CREAT and IPC_EXCL
50                    flags.*/
51          opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
52      }
53      /*Get the number of semaphores for this set.*/
54      printf("\nEnter the number of\n");
55      printf("desired semaphores for\n");
56      printf("this set (25 max) = ");
57      scanf("%d", &nsems);

58      /*Check the entry.*/
59      printf("\nNsems = %d\n", nsems);
```

```
60      /*Call the semget system call.*/
61      semid = semget(key, nsems, opperm_flags);

62      /*Perform if the call is unsuccessful.*/
63      if(semid == -1)
64      {
65          printf("The semget system call failed!\n");
66          printf("The error number = %d\n", errno);
67      }
68      /*Return the semid upon successful completion.*/
69      else
70          printf("\nThe semid = %d\n", semid);
71      exit(0);
72  }
```

The program begins by prompting for a hexadecimal **key**, an octal operation permissions code, and the control command combinations (flags) that are selected from a menu (lines 15-32). All possible combinations are allowed even though they might not be viable. From this you can observe illegal combination errors.

Next, the menu selection for the flags is combined with the operation permissions, and the result is stored in the **opperm_flags** variable (lines 36-52).

Then the number of semaphores for the set is requested (lines 53-57), and its value is stored in **nsems**.

The system call is made next, and the result is stored in the **semid** variable (lines 60, 61).

Since the **semid** variable now contains a valid semaphore set identifier or the error code (–1), it is tested to see if an error occurred (line 63). If **semid** equals –1, a message indicates that an error resulted and the external **errno** variable is displayed (lines 65, 66). Remember that the external **errno** variable is only set when a system call fails and should only be tested immediately following system calls.

If no error occurred, the returned semaphore set identifier is displayed (line 70).

4.3.3 Controlling Semaphores

This section contains a detailed description of using the **semctl**(2) system call along with an sample program that exercises all of its capabilities.

Using semctl

The synopsis found in the **semctl**(2) man page is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl (semid, semnum, cmd, arg)
int semid, cmd;
int semnum;
union semun
{
        int val;
        struct semid_ds *bu;
        ushort array[];
} arg;
```

The **semctl**(2) system call requires four arguments to be passed to it: **semid**, **semnum**, **cmd**, and **arg**. It returns an integer value. A description of the four arguments follows:

semid Must be a valid, non-negative, integer value that has already been created by using the **semget**(2) system call.

semnum Selects a semaphore within a set by its number. This relates to array (atomically performed) operations on the set. When a set of semaphores is created, the first semaphore is number 0, and the last semaphore has the number of one less than the total in the set.

cmd Can be replaced by one of the following control commands (flags):

GETVAL Return the value of a single semaphore within a semaphore set.

SETVAL	Set the value of a single semaphore within a semaphore set.
GETPID	Return the Process Identifier (PID) of the process that performed the last operation on the semaphore within a semaphore set.
GETNCNT	Return the number of processes waiting for the value of a particular semaphore to become greater than its current value.
GETZCNT	Return the number of processes waiting for the value of a particular semaphore to be equal to zero.
GETALL	Return the values for all semaphores in a semaphore set.
SETALL	Set all semaphore values in a semaphore set.
IPC_STAT	Return the status information contained in the associated data structure for the specified **semid**, and place it in the data structure pointed to by the ***buf** pointer in the user memory area; **arg.buf** is the union member that contains the value of **buf**.
IPC_SET	For the specified semaphore set (**semid**), set the effective user/group identification and operation permissions.
IPC_RMID	Remove the specified (**semid**) semaphore set along with its associated data structure. A process must have an effective user identification of OWNER/CREATOR or super-user to perform an IPC_SET or IPC_RMID control command. Read/alter permission is required as applicable for the other control commands.

arg Passes the system call the appropriate union member for the control command to be performed:

- **arg.val**

- **arg.buf**
- **arg.array**

The details of this system call are discussed in the following sample program. If you have problems understanding the logic in this program, read "Using **semget**" earlier in this chapter.

Sample Program

The sample program in this section is a menu-driven program that allows you to exercise all possible combinations of the **semctl**(2) system call.

Lines 5-9 include the required header files as specified by the **semctl**(2) man page.

Variables declared for this program and their purpose are as follows:

semid_ds	Receives the specified semaphore set identifier's data structure when an IPC_STAT control command is performed.
c	Receives the input values from the **scanf**(3S) function, (line 117) when performing a SETALL control command.
i	A counter to increment through the union **arg.array** when displaying the semaphore values for a GETALL (lines 97-99) control command, and when initializing the **arg.array** when performing a SETALL (lines 115-119) control command.
length	A variable to test for the number of semaphores in a set against the **i** counter variable (lines 97, 115).
uid	Stores the IPC_SET value for the effective user identification.
gid	Stores the IPC_SET value for the effective group identification.
mode	Stores the IPC_SET value for the operation permissions.
rtrn	Stores the return integer from the system call which depends upon the control command or a –1 when unsuccessful.
semid	Stores and passes the semaphore set identifier to the system call.

semnum Stores and passes the semaphore number to the system call.

cmd Stores the code for the desired control command so that subsequent processing can be performed on it.

choice Determines which member (**uid, gid, mode**) is to be changed for the IPC_SET control command.

arg.val Passes the system call a value to set (SETVAL) or stores (GETVAL) a value returned from the system call for a single semaphore (union member).

arg.buf A pointer passed to the system call which locates the data structure in the user memory area where the IPC_STAT control command is to place its return values, or where the IPC_SET command gets the values to set (union member).

arg.array Stores the set of semaphore values when getting (GETALL) or initializing (SETALL) (union member).

The sample program for the **semctl**(2) system call is shown in the following example:

```
1  /*This is a program to illustrate
2  **the semaphore control, semctl(),
3  **system call capabilities.
4  */

5  /*Include necessary header files.*/
6  #include    <stdio.h>
7  #include    <sys/types.h>
8  #include    <sys/ipc.h>
9  #include    <sys/sem.h>

10  /*Start of main C language program*/
11  main()
12  {
13      extern int errno;
14      struct semid_ds semid_ds;
15      int c, i, length;
16      int uid, gid, mode;
17      int retrn, semid, semnum, cmd, choice;
18      union semun  {
19          int val;
20          struct semid_ds *buf;
```

```
21          ushort array[25];
22      } arg;

23      /*Initialize the data structure pointer.*/
24      arg.buf = &semid_ds;
```

The **arg** union (lines 18-22) serves three purposes. The compiler allocates enough storage to hold its largest member. The program can then use the union as any member by referencing union members as if they were regular structure members. Note that the array is declared to have 25 elements (0 through 24). This number corresponds to the maximum number of semaphores allowed per set (SEMMSL), a system tunable parameter.

The next important program aspect to observe is that although the ***buf** pointer member (**arg.buf**) of the union is declared to be a pointer to a data structure of the **semid_ds** type, it must also be initialized to contain the address of the user memory area data structure (line 24). Because of the way this program is written, the pointer does not need to be reinitialized later. If it was used to increment through the array, it would need to be reinitialized just before calling the system call.

```
25      /*Enter the semaphore ID.*/
26      printf("Enter the semid = ");
27      scanf("%d", &semid);

28      /*Choose the desired command.*/
29      printf("\nEnter the number for\n");
30      printf("the desired cmd:\n");
31      printf("GETVAL      =  1\n");
32      printf("SETVAL      =  2\n");
33      printf("GETPID      =  3\n");
34      printf("GETNCNT     =  4\n");
35      printf("GETZCNT     =  5\n");
36      printf("GETALL      =  6\n");
37      printf("SETALL      =  7\n");
38      printf("IPC_STAT    =  8\n");
39      printf("IPC_SET     =  9\n");
40      printf("IPC_RMID    =  10\n");
41      printf("Entry       =  ");
42      scanf("%d", &cmd);

43      /*Check entries.*/
44      printf ("\nsemid =%d, cmd = %d\n\n",
```

```
45          semid, cmd);

46      /*Set the command and do the call.*/
47      switch (cmd)
48      {

49      case 1: /*Get a specified value.*/
50          printf("\nEnter the semnum = ");
51          scanf("%d", &semnum);
52          /*Do the system call.*/
53          retrn = semctl(semid, semnum, GETVAL, 0);
54          printf("\nThe semval = %d\n", retrn);
55          break;
56      case 2: /*Set a specified value.*/
57          printf("\nEnter the semnum = ");
58          scanf("%d", &semnum);
59          printf("\nEnter the value = ");
60          scanf("%d", &arg.val);
61          /*Do the system call.*/
62          retrn = semctl(semid, semnum, SETVAL, arg.val);
63          break;
64      case 3: /*Get the process ID.*/
65          retrn = semctl(semid, 0, GETPID, 0);
66          printf("\nThe sempid = %d\n", retrn);
67          break;
68      case 4: /*Get the number of processes
69            waiting for the semaphore to
70            become greater than its current
71                    value.*/
72          printf("\nEnter the semnum = ");
73          scanf("%d", &semnum);
74          /*Do the system call.*/
75          retrn = semctl(semid, semnum, GETNCNT, 0);
76          printf("\nThe semncnt = %d", retrn);
77          break;

78      case 5: /*Get the number of processes
79            waiting for the semaphore
80                    value to become zero.*/
81          printf("\nEnter the semnum = ");
82          scanf("%d", &semnum);
83          /*Do the system call.*/
```

```
84          retrn = semctl(semid, semnum, GETZCNT, 0);
85          printf("\nThe semzcnt = %d", retrn);
86          break;

87      case 6: /*Get all of the semaphores.*/
88          /*Get the number of semaphores in
89            the semaphore set.*/
90          retrn = semctl(semid, 0, IPC_STAT, arg.buf);
91          length = arg.buf->sem_nsems;
92          if(retrn == -1)
93              goto ERROR;
94          /*Get and print all semaphores in the
95            specified set.*/
96          retrn = semctl(semid, 0, GETALL, arg.array);
97          for (i = 0; i < length; i++)
98          {
99              printf("%d", arg.array[i]);
100             /*Separate each
101                semaphore.*/
102              printf("%c", ' ');
103         }
104         break;

105     case 7: /*Set all semaphores in the set.*/
106         /*Get the number of semaphores in
107           the set.*/
108         retrn = semctl(semid, 0, IPC_STAT, arg.buf);
109         length = arg.buf->sem_nsems;
110         printf("Length = %d\n", length);
111         if(retrn == -1)
112             goto ERROR;
113         /*Set the semaphore set values.*/
114         printf("\nEnter each value:\n");
115         for(i = 0; i < length ; i++)
116         {
117             scanf("%d", &c);
118             arg.array[i] = c;
119         }
120         /*Do the system call.*/
121         retrn = semctl(semid, 0, SETALL, arg.array);
122         break;
```

```
123       case 8: /*Get the status for the semaphore set.*/
125           /*Get and print the current status values.*/
127           retrn = semctl(semid, 0, IPC_STAT, arg.buf);
128           printf ("\nThe USER ID = %d\n",
129               arg.buf->sem_perm.uid);
130           printf ("The GROUP ID = %d\n",
131               arg.buf->sem_perm.gid);
132           printf ("The operation permissions = 0%o\n",
133               arg.buf->sem_perm.mode);
134           printf ("The number of semaphores in set = %d\n",
135               arg.buf->sem_nsems);
136           printf ("The last semop time = %d\n",
137               arg.buf->sem_otime);

138           printf ("The last change time  = %d\n",
139               arg.buf->sem_ctime);
140           break;

141       case 9:     /*Select and change the desired
142                       member of the data structure.*/
143           /*Get the current status values.*/
144           retrn = semctl(semid, 0, IPC_STAT, arg.buf);
145           if(retrn == -1)
146               goto ERROR;
147           /*Select the member to change.*/
148           printf("\nEnter the number for the\n");
149           printf("member to be changed:\n");
150           printf("sem_perm.uid    = 1\n");
151           printf("sem_perm.gid    = 2\n");
152           printf("sem_perm.mode   = 3\n");
153           printf("Entry           = ");
154           scanf("%d", &choice);
155           switch(choice){

156           case 1: /*Change the user ID.*/
157               printf("\nEnter USER ID = ");
158               scanf ("%d", &uid);
159               arg.buf->sem_perm.uid = uid;
160               printf("\nUSER ID = %d\n",
161                   arg.buf->sem_perm.uid);
162               break;
```

```
163          case 2: /*Change the group ID.*/
164              printf("\nEnter GROUP ID = ");
165              scanf("%d", &gid);
166              arg.buf->sem_perm.gid = gid;
167              printf("\nGROUP ID = %d\n",
168                  arg.buf->sem_perm.gid);
169              break;

170          case 3: /*Change the mode portion of
171                the operation
172                            permissions.*/
173              printf("\nEnter MODE = ");
174              scanf("%o", &mode);
175              arg.buf->sem_perm.mode = mode;
176              printf("\nMODE = 0%o\n",
177                  arg.buf->sem_perm.mode);
178              break;
179          }
180          /*Do the change.*/
181          retrn = semctl(semid, 0, IPC_SET, arg.buf);
182          break;
183      case 10:    /*Remove the semid along with its
184                    data structure.*/
185          retrn = semctl(semid, 0, IPC_RMID, 0);
186      }
187      /*Perform if the call is unsuccessful.*/
188      if(retrn == -1)
189      {
190  ERROR:
191          printf ("\n\nThe semctl system call failed!\n");
192          printf ("The error number = %d\n", errno);
193          exit(0);
194      }
195      printf ("\n\nThe semctl system call was successful\n");
196      printf ("for semid = %d\n", semid);
197      exit (0);
198  }
```

First, the program prompts for a valid semaphore set identifier, which is stored in the **semid** variable (lines 25-27). This is required for all **semctl**(2) system calls.

Then, the code for the desired control command must be entered (lines 28-42), and the code is stored in the **cmd** variable. The code is tested to determine the control command for subsequent processing.

If the GETVAL control command is selected (code 1), a message prompting for a semaphore number is displayed (lines 49, 50). When it is entered, it is stored in **semnum** variable (line 51). Then, the system call is performed, and the semaphore value is displayed (lines 52-55). If the system call is successful, a message indicates this along with the semaphore set identifier used (lines 195, 196); if the system call is unsuccessful, an error message is displayed along with the value of the external **errno** variable (lines 191-193).

If the SETVAL control command is selected (code 2), a message prompting for a semaphore number is displayed (lines 56, 57). When it is entered, it is stored in the **semnum** variable (line 58). Next, a message prompts for the value to which the semaphore is to be set, and it is stored as the **arg.val** member of the union (lines 59, 60). Then, the system call is performed (lines 61, 63). Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the GETPID control command is selected (code 3), the system call is made immediately since all required arguments are known (lines 64-67), and the PID of the process performing the last operation is displayed. Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the GETNCNT control command is selected (code 4), a message prompting for a semaphore number is displayed (lines 68-72). When entered, it is stored in the **semnum** variable (line 73). Then, the **semctl()** system call is performed, and the number of processes waiting for the semaphore to become greater than its current value is displayed (lines 74-77). Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the GETZCNT control command is selected (code 5), a message prompting for a semaphore number is displayed (lines 78-81). When it is entered, it is stored in the **semnum** variable (line 82). Then the **semctl()** system call is performed, and the number of processes waiting for the semaphore value to become equal to zero is displayed (lines 83, 86). Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the GETALL control command is selected (code 6), the program first performs an IPC_STAT control command to determine the number of semaphores in the set (lines 88-93). The length variable is set to the number of semaphores in the set (line 91). Next, the **semctl()** system call is made and,

upon success, the **arg.array** union member contains the values of the semaphore set (line 96). Now, a loop is entered which displays each element of the **arg.array** from zero to one less than the value of *length* (lines 97-103). The semaphores in the set are displayed on a single line, each separated by a space. Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the SETALL control command is selected (code 7), the program first performs an IPC_STAT control command to determine the number of semaphores in the set (lines 106-108). The length variable is set to the number of semaphores in the set (line 109). Next, the program prompts for the values to be set and enters a loop which takes values from the keyboard and initializes the **arg.array** union member to contain the desired values of the semaphore set (lines 113-119). The loop puts the first entry into the array position for semaphore number zero and ends when the semaphore number that is filled in the array equals one less than the value of length. The **semctl()** system call is then made (lines 120-122). Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the IPC_STAT control command is selected (code 8), the **semctl()** system call is performed (line 127), and the status information returned is printed out (lines 128-139); only the members that can be set are printed out in this program. Note that if the system call is unsuccessful, the status information of the last successful one is printed out. In addition, an error message is displayed, and the **errno** variable is printed out (lines 191, 192).

If the IPC_SET control command is selected (code 9), the program gets the current status information for the semaphore set identifier specified (lines 143-146). This is necessary because this sample program provides for changing only one member at a time, and the **semctl**(2) system call changes all of them. Also, if an invalid value happened to be stored in the user memory area for one of these members, it would cause repetitive failures for this control command until corrected. The next thing the program does is to prompt for a code corresponding to the member to be changed (lines 147-153). This code is stored in the **choice** variable (line 154). Now, depending upon the member picked, the program prompts for the new value (lines 155-178). The value is placed at the address of the appropriate member in the user memory area data structure, and the system call is made (line 181). Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the IPC_RMID control command (code 10) is selected, the system call is performed (lines 183-185). The **semid** along with its associated data structure and semaphore set is removed from the operating system. Depending upon success or failure, the program returns the same messages as other control commands.

4.3.4 Operations on Semaphores

This section contains a detailed description of using the **semop**(2) system call along with a sample program that exercises all of its capabilities.

Using semop

The synopsis found in the **semop**(2) man page is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop (semid, sops, nsops)
int semid;
struct sembuf **sops;
unsigned nsops;
```

The **semop**(2) system call requires three arguments to be passed to it: **semid**, **sops**, and **nsops**. It returns an integer value. Upon successful completion, a zero value is returned and when unsuccessful it returns a –1. Here are the three arguments:

semid Must be a non-negative integer value returned from the **semget**(2) system call.

sops A pointer to an array of structures in the user memory area that contains the following for each semaphore to be changed:

- The semaphore number
- The operation to be performed
- The control command (flags)

The ****sops** declaration means that a pointer can be initialized to the address of the array, or the array name can be used since it is the address of the first element of the array. **Sembuf** is the *tag* name of the data structure used as the template for the

structure members in the array. It is located in the **#include <sys/sem.h>** header file.

nsops Specifies the length of the array (the number of structures in the array). The maximum **size** of this array is determined by the SEMOPM system tunable parameter. Therefore, a maximum of SEMOPM operations can be performed for each **semop**(2) system call.

The semaphore number determines the particular semaphore within the set on which the operation is to be performed. The specific operation to be performed is determined by these rules:

- A positive integer value means to increment the semaphore value by its value.
- A negative integer value means to decrement the semaphore value by its value.
- A value of zero means to test if the semaphore is equal to zero.

The following operation commands (flags) can be used:

NOWAIT If the requested operations on a semaphore set cannot be performed, you can choose to have the process *block* in the system call until all operations succeed. You can also choose to have the system call fail immediately. The system call will fail immediately if NOWAIT is set.

UNDO UNDO specifies that the cumulative effect of all semaphore operations a specific process performs on a specific semaphore set will be "remembered" in the kernel. When that process exits, these remembered values are used to "undo" the effects of that process on that semaphore set.

Sample Program

The sample program in this section is a menu-driven program that exercises all possible combinations of the **semop**(2) system call.

Lines 5-9 include the required header files as specified by the **shmop**(2) man page. The variables declared for this program are as follows:

sembuf[10] An array buffer (line 14) to contain a maximum of ten **sembuf** type structures; ten equals SEMOPM, the maximum number of operations on a semaphore set for each **semop**(2) system call.

***sops** A pointer (line 14) to **sembuf**[10] for the system call and for accessing the structure members within the array.

rtrn Stores the return values from the system call.

flags Stores the code of the IPC_NOWAIT or SEM_UNDO flags for the **semop**(2) system call (line 60).

i A counter (line 32) for initializing the structure members in the array, and used to print out each structure in the array (line 79).

nsops Specifies the number of semaphore operations for the system call—must be less than or equal to SEMOPM.

semid Stores the desired semaphore set identifier for the system call.

The sample program for the **semop**(2) system call is shown here:

```
1  /*This is a program to illustrate
2  **the semaphore operations, semop(),
3  **system call capabilities.
4  */

5  /*Include necessary header files.*/
6  #include    <stdio.h>
7  #include    <sys/types.h>
8  #include    <sys/ipc.h>
9  #include    <sys/sem.h>
10  /*Start of main C language program*/
11  main()
12  {
13      extern int errno;
14      struct sembuf sembuf[10], *sops;
15      char string[];
16      int retrn, flags, sem_num, i, semid;
17      unsigned nsops;
18      sops = sembuf; /*Pointer to array sembuf.*/
```

```
19      /*Enter the semaphore ID.*/
20      printf("\nEnter the semid of\n");
21      printf("the semaphore set to\n");
22      printf("be operated on = ");
23      scanf("%d", &semid);
24      printf("\nsemid = %d", semid);
25      /*Enter the number of operations.*/
26      printf("\nEnter the number of semaphore\n");
27      printf("operations for this set = ");
28      scanf("%d", &nsops);
29      printf("\nnosops = %d", nsops);

30      /*Initialize the array for the
31        number of operations to be performed.*/
32      for(i = 0; i < nsops; i++, sops++)
33      {

34          /*This determines the semaphore in
35            the semaphore set.*/
36          printf("\nEnter the semaphore\n");
37          printf("number (sem_num) = ");
38          scanf("%d", &sem_num);
39          sops->sem_num = sem_num;
40          printf("\nThe sem_num = %d", sops->sem_num);

41          /*Enter a (-)number to decrement,
42            an unsigned number (no +) to increment,
43            or zero to test for zero.  These values
44            are entered into a string and converted
45            to integer values.*/
46          printf("\nEnter the operation for\n");
47          printf("the semaphore (sem_op) = ");
48          scanf("%s", string);
49          sops->sem_op = atoi(string);
50          printf("\nsem_op = %d\n", sops->sem_op);
51          /*Specify the desired flags.*/
52          printf("\nEnter the corresponding\n");
53          printf("number for the desired\n");
54          printf("flags:\n");
55          printf("No flags                    = 0\n");
56          printf("IPC_NOWAIT                  = 1\n");
57          printf("SEM_UNDO                    = 2\n");
```

```
58          printf("IPC_NOWAIT and SEM_UNDO    = 3\n");
59          printf("              Flags          = ");
60          scanf("%d", &flags);

61          switch(flags)
62          {
63          case 0:
64              sops->sem_flg = 0;
65              break;
66          case 1:
67              sops->sem_flg = IPC_NOWAIT;
68              break;
69          case 2:
70              sops->sem_flg = SEM_UNDO;
71              break;
72          case 3:
73              sops->sem_flg = IPC_NOWAIT | SEM_UNDO;
74              break;
75          }
76          printf("\nFlags = 0%o\n", sops->sem_flg);
77      }
78      /*Print out each structure in the array.*/
79      for(i = 0; i < nsops; i++)
80      {
81          printf("\nsem_num = %d\n", sembuf[i].sem_num);
82          printf("sem_op = %d\n", sembuf[i].sem_op);
83          printf("sem_flg = %o\n", sembuf[i].sem_flg);
84          printf("%c", ' ');
85      }

86      sops = sembuf; /*Reset the pointer to
87                       sembuf[0].*/

88      /*Do the semop system call.*/
89      retrn = semop(semid, sops, nsops);
90      if(retrn == -1)  {
91          printf("\nSemop failed.  ");
92          printf("Error = %d\n", errno);
93      }
94      else {
95          printf ("\nSemop was successful\n");
96          printf("for semid = %d\n", semid);
```

```
97              printf("Value returned = %d\n", retrn);
98          }
99      }
```

First, the program prompts for a semaphore set identifier that the system call is to perform operations on (lines 19-22). The semaphore id is stored in the **semid** variable (line 23).

A message is displayed requesting the number of operations to be performed on this set (lines 25-27). The number of operations is stored in the **nsops** variable (line 28).

Next, a loop is entered to initialize the array of structures (lines 30-77). The semaphore number, operation, and operation command (flags) are entered for each structure in the array. The number of structures equals the number of semaphore operations (**nsops**) to be performed for the system call, so **nsops** is tested against the **i** counter for loop control. Note that **sops** is used as a pointer to each element (structure) in the array, and **sops** is incremented just like **i**. **sops** is then used to point to each member in the structure for setting them.

After the array is initialized, all of its elements are printed out for feedback (lines 78-85).

The **sops** pointer is set to the address of the array (lines 86, 87). **sembuf** could be used directly, if desired, instead of **sops** in the system call.

The system call is made (line 89) and, depending upon success or failure, a corresponding message is displayed. The results of the operation(s) can be viewed by using the **semctl**() GETALL control command.

4.4 Shared Memory

Sharing of memory provides the fastest means of exchanging data between processes. The shared memory type of IPC allows two or more processes (executing programs) to share memory and, consequently, the data contained there. This is done by allowing processes to set up access to a common virtual memory address space.

A process initially creates a shared memory segment facility with the **shmget**(2) system call. Upon creation, this process sets the overall operation permissions for the shared memory segment facility, sets its size in bytes, and can specify that the shared memory segment is for reference only (read-only) upon attachment. If the memory segment is not specified to be for reference only, all other processes with appropriate operation permissions can

read from or write to the memory segment.

There are two operations that can be performed on a shared memory segment:

shmat(2)	Shared memory attach
shmdt(2)	Shared memory detach

Shared memory attach allows processes to associate themselves with the shared memory segment if they have permission. They can then read or write as allowed.

Shared memory detach allows processes to disassociate themselves from a shared memory segment. Therefore, they lose the ability to read from or write to the shared memory segment.

The original owner/creator of a shared memory segment can relinquish ownership to another process using the **shmctl**(2) system call. However, the creating process remains the creator until the facility is removed or the system is reinitialized. Other processes with permission can perform other functions on the shared memory segment using the **shmctl**(2) system call.

System calls, which are documented in the *Reference Manual*, make these shared memory capabilities available to processes. The calling process passes arguments to a system call, and the system call either successfully or unsuccessfully performs its function. If the system call is successful, it performs its function and returns the appropriate information. Otherwise, a known error code (–1) is returned to the process, and the external variable **errno** is set accordingly.

4.4.1 Using Shared Memory

The sharing of memory between processes occurs on a virtual segment basis. There is one and only one instance of an individual shared memory segment existing in the operating system at any one time.

Before sharing of memory can be realized, a uniquely identified shared memory segment and data structure must be created. The unique identifier created is called the shared memory identifier (**shmid**). It is used to identify or reference the associated data structure. The data structure includes the following for each shared memory segment:

- Operation permissions
- Segment size
- Segment descriptor
- Process identification performing last operation
- Process identification of creator
- Current number of processes attached
- Number of processes attached that are currently in memory (for example, not swapped out
- Last attach time
- Last change time

The data structure definition for the shared memory segment data structure is located in the **/usr/include/sys/shm.h** header file. It is shown here:

```
/*
**    There is a shared mem id data structure for
**    each segment in the system.
*/

struct shmid_ds {
    struct ipc_perm  shm_perm;    /* operation permission struct */
    int              shm_segsz;   /* segment size */
    struct region    *shm_reg;    /* ptr to region structure */
    char             pad[4];      /* for swap compatibility */
    ushort           shm_lpid;    /* pid of last shmop */
    ushort           shm_cpid;    /* pid of creator */
    ushort           shm_nattch;  /* used only for shminfo */
    ushort           shm_cnattch; /* used only for shminfo */
    time_t           shm_atime;   /* last shmat time */
    time_t           shm_dtime;   /* last shmdt time */
    time_t           shm_ctime;   /* last change time */
};
```

Note that the **shm_perm** member of this structure uses **ipc_perm** as a template. The breakout for the operation permissions data structure is shown in the **ipc_perm** data structure, earlier in this chapter.

The **ipc_perm** data structure is the same for all IPC facilities, and it is located in the **#include <sys/ipc.h>** header file. It is shown in the introduction section of "Messages." Table 4-5 shows the shared memory state information.

Table 4-5
Shared Memory State Information

Shared Memory States			
Lock Bit	**Swap Bit**	**Allocated Bit**	**Implied State**
0	0	0	Unallocated Segment
0	0	1	Incore
0	1	0	Unused
0	1	1	On Disk
1	0	1	Unused
1	1	0	Unused
1	0	0	Unused
1	1	1	Unused

The implied states of Table 4-5 are as follows:

Unallocated Segment The segment associated with this segment descriptor has not been allocated for use.

Incore The shared segment associated with this descriptor has been allocated for use. Therefore, the segment does exist and is currently resident in memory.

On Disk The shared segment associated with this segment descriptor is currently resident on the swap device.

Unused This state is currently unused and should never be encountered by the normal user in shared memory handling.

The **shmget**(2) system call performs one of two tasks when only the IPC_CREAT flag is set in the **shmflg** argument that it receives:

- Get a new **shmid** and create an associated shared memory segment data structure for it.
- Return an existing **shmid** that already has an associated shared memory segment data structure.

The task performed is determined by the value of the **key** argument passed to the **shmget**(2) system call. For the first task, if the **key** is not already in use for an existing **shmid**, a new **shmid** is returned with an associated shared memory segment data structure created for it provided no system tunable parameters would be exceeded.

There is also a provision for specifying a **key** of value zero which is known as the private **key** (IPC_PRIVATE = 0); when specified, a new **shmid** is always returned with an associated shared memory segment data structure created for it unless a system tunable parameter would be exceeded. When the **ipcs** command is performed, the KEY field for the **shmid** is all zeros.

For the second task, if a **shmid** exists for the **key** specified, the value of the existing **shmid** is returned. If it is not desired to have an existing **shmid** returned, a control command (IPC_EXCL) can be specified (set) in the **shmflg** argument passed to the system call. The details of using this system call are discussed in "Using **shmget**" later in this chapter.

When performing the first task, the process that calls **shmget** becomes the owner/creator, and the associated data structure is initialized accordingly. Remember, ownership can be changed, but the creating process always remains the creator (see "Controlling Shared Memory" later in this chapter). The creator of the shared memory segment also determines the initial operation permissions for it.

Once a shared memory segment data structure is created, shared memory segment operations [**shmop**()] and control [**shmctl**(2)] can be used.

Shared memory segment operations consist of attaching and detaching shared memory segments. System calls are provided for each of these operations. They are **shmat**(2) and **shmdt**(2). Refer to "Operations for Shared Memory" later in this chapter for details.

Shared memory segment control is done by using the **shmctl**(2) system call. It permits you to control the shared memory facility in the following ways:

- Determine the associated data structure status for a shared memory segment (**shmid**).
- Change operation permissions for a shared memory segment.
- Remove a particular **shmid** from the operating system along with its associated shared memory segment data structure.

Refer to "Controlling Shared Memory" later in this chapter for details.

4.4.2 Getting Shared Memory Segments

This section explains how to use the **shmget**(2) system call and provides a sample program illustrating its use.

Using shmget

The synopsis found in the **shmget**(2) man page is as follows:

```
#include  <sys/types.h>
#include  <sys/ipc.h>
#include  <sys/shm.h>

int  shmget (key, size, shmflg)
key_t  key;
int size, shmflg;
```

Upon successful completion, the integer returned from this function is the shared memory identifier (**shmid**).

A new **shmid** with an associated shared memory data structure is provided if either of these conditions is met:

- **key** is equal to IPC_PRIVATE.
- **key** is passed a unique integer, and **shmflg** ANDed with IPC_CREAT is TRUE.

The value passed to the **shmflg** argument must be an integer value and must specify the following:

- Operation permissions
- Control fields (commands)

Operation permissions determine the read/write attributes for the owner/group/other classes of operations. Table 4-6 reflects the numeric values (expressed in octal notation) for the valid operation permissions codes.

Table 4-6
shmflg Operation Permissions Codes

Operation Permissions	Octal Value
Read by Owner	00400
Write by Owner	00200
Read by Group	00040
Write by Group	00020
Read by Others	00004
Write by Others	00002

A specific value is derived by adding the values for the operation permissions desired. That is, if you want read by user and read/write by others, the code value would be 00406 (00400 plus 00004 plus 00002). There are constants located in the **shm.h** header file that can be used for the owner. They are as follows:

```
SHM_R  0400
SHM_W  0200
```

Control commands are predefined constants (represented by all uppercase letters). Table 4-7 contains the names and values of the constants that apply to the **shmget**() system call. These constants are also referred to as flags and are defined in the **ipc.h** header file.

Table 4-7
shmget Control Commands (Flags)

Control Command	Octal Value
IPC_CREAT	0001000
IPC_EXCL	0002000

The value for **shmflg** is, therefore, a combination of operation permissions and control commands. The desired flag(s) can be specified after determining the value for the operation permissions, as previously described. This is accomplished with a bitwise OR (|) of the flags with the operation permissions.

As specified by the **shmget**(2) man page, the success or failure of this system call depends upon the argument values for **key**, **size**, and **shmflg** or system tunable parameters. The system call will attempt to return a new **shmid** if one of the following conditions is true:

- **key** is equal to IPC_PRIVATE (0).
- **key** does not already have a **shmid** associated with it, and (**shmflg** & IPC_CREAT) is true (not zero).

Exceeding the SHMMNI system tunable parameter always causes a failure. The SHMMNI system tunable parameter determines the maximum number of unique shared memory segments (**shmids**) in the operating system.

IPC_EXCL is another control command used in conjunction with IPC_CREAT. If IPC_EXCL is set, the **shmget**() system call fails if, and only if, a **shmid** exists for the specified **key**. This is necessary to prevent the process from thinking that it has received a new (unique) **shmid** when it has not. If the system call is successful, a unique **shmid** is returned when both IPC_CREAT and IPC_EXCL are specified. Any value for **shmflg** returns a new **shmid** if the **key** equals zero (IPC_PRIVATE).

The system call will fail if the value for the **size** argument is less than SHMMIN or greater than SHMMAX. These tunable parameters specify the minimum and maximum shared memory segment **sizes**.

Refer to the **shmget**(2) manual page for more information. The specific failure conditions with error names are contained there also.

Sample Program

The sample program in this section is a menu-driven program that exercises all possible combinations of the **shmget**(2) system.

Lines 4-7 include the required header files as specified by the **shmget**(2) man page. The variables declared for this program are as follows:

key	The value for the desired **key**.
opperm	Stores the desired operation permissions.
flags	Stores the desired control commands (flags).
opperm_flags	Stores the combination from the logical ORing of the **opperm** and **flags** variables; it is then used in the system call to pass the **shmflg** argument.
shmid	The shared memory identification number for a successful system call or the error code (–1) for an unsuccessful one.
size	Specifies the shared memory segment size.

The sample program for the **shmget**(2) system call is shown here:

```
1  /*This is a program to illustrate
2  *the shared memory get, shmget(),
3  *system call capabilities.
4  */

5  #include    <sys/types.h>
6  #include    <sys/ipc.h>
7  #include    <sys/shm.h>
8  #include    <errno.h>

9  /*Start of main C language program*/
10  main()
11  {
12       key_t key;              /*declare as long integer*/
13       int opperm, flags;
14       int shmid, size, opperm_flags;
15       /*Enter the desired key*/
16       printf("Enter the desired key in hex = ");
17       scanf("%x", &key);
```

```
18      /*Enter the desired octal operation
19        permissions.*/
20      printf("\nEnter the operation\n");
21      printf("permissions in octal = ");
22      scanf("%o", &opperm);
22      /*Set the desired flags.*/
23      printf("\nEnter corresponding number to\n");
24      printf("set the desired flags:\n");
25      printf("No flags                    = 0\n");
26      printf("IPC_CREAT                   = 1\n");
27      printf("IPC_EXCL                    = 2\n");
28      printf("IPC_CREAT and IPC_EXCL      = 3\n");
29      printf("            Flags           = ");
30      /*Get the flag(s) to be set.*/
31      scanf("%d", &flags);

32      /*Check the values.*/
33      printf ("\nkey =0x%x, opperm = 0%o, flags = 0%o\n",
34          key, opperm, flags);

35      /*Incorporate the control fields (flags) with
36        the operation permissions*/
37      switch (flags)
38      {
39      case 0:     /*No flags are to be set.*/
40          opperm_flags = (opperm | 0);
41          break;
42      case 1:     /*Set the IPC_CREAT flag.*/
43          opperm_flags = (opperm | IPC_CREAT);
44          break;
45      case 2:     /*Set the IPC_EXCL flag.*/
46          opperm_flags = (opperm | IPC_EXCL);
47          break;
48      case 3:     /*Set the IPC_CREAT and IPC_EXCL flags.*/
49          opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
50      }
51      /*Get the size of the segment in bytes.*/
52      printf ("\nEnter the segment");
53      printf ("\nsize in bytes = ");
54      scanf ("%d", &size);

55      /*Call the shmget system call.*/
```

```
56      shmid = shmget (key, size, opperm_flags);

57      /*Perform if the call is unsuccessful.*/
58      if(shmid == -1)
59      {
60          printf ("\nThe shmget system call failed!\n");
61          printf ("The error number = %d\n", errno);
62      }
63      /*Return the shmid upon successful completion.*/
64      else
65          printf ("\nThe shmid = %d\n", shmid);
66      exit(0);
67  }
```

The program begins by prompting for a hexadecimal **key**, an octal operation permissions code, and finally for the control command combinations (flags) which are selected from a menu (lines 14-31). All possible combinations are allowed even though they might not be viable. This allows you to observe the errors for illegal combinations.

Next, the menu selection for the flags is combined with the operation permissions, and the result is stored in the **opperm_flags** variable (lines 35-50).

A display then prompts for the **size** of the shared memory segment, and it is stored in the **size** variable (lines 51-54).

The system call is made next, and the result is stored in the **shmid** variable (line 56).

Since the **shmid** variable now contains a valid message queue identifier or the error code (–1), it is tested to see if an error occurred (line 58). If **shmid** equals –1, a message indicates that an error resulted and the external **errno** variable is displayed (lines 60, 61).

If no error occurred, the returned shared memory segment identifier is displayed (line 65).

4.4.3 Controlling Shared Memory

This section explains how to use the **shmctl**(2) system call and provides a sample program that allows all of its capabilities to be exercised.

Using shmctl

The synopsis found in the **shmctl**(2) man page is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl (shmid, cmd, buf)
int shmid, cmd;
struct shmid_ds *buf;
```

The **shmctl**(2) system call requires three arguments to be passed to it: **shmid**, **cmd**, and **buf**. Upon successful completion, a zero value is returned. When unsuccessful, **shmctl**() returns a –1.

The **shmid** variable must be a valid, non-negative, integer value and must have already been created by the **shmget**(2) system call.

The **cmd** argument can be replaced by one of following control commands (flags):

IPC_STAT	Return the status information contained in the associated data structure for the specified **shmid** and place it in the data structure pointed to by the ***buf** pointer in the user memory area.
IPC_SET	For the specified **shmid**, set the effective user and group identification, and operation permissions.
IPC_RMID	Remove the specified **shmid** along with its associated shared memory segment data structure.

A process must have an effective user identification of OWNER/CREATOR or superuser to perform an IPC_SET or IPC_RMID control command. A process must have read permission to perform the IPC_STAT control command.

The details of this system call are shown in the sample program. If you have problems understanding this program, read "Using **shmget**" earlier in this chapter for more detail.

Sample Program

The sample program in this section is a menu-driven program that exercises all possible combinations of the **shmctl**(2) system call.

Lines 5-9 include the required header files as specified by the **shmctl**(2) man page.

The variables declared for this program and their purposes are as follows:

uid	Stores the IPC_SET value for the effective user identification.
gid	Stores the IPC_SET value for the effective group identification.
mode	Stores the IPC_SET value for the operation permissions.
rtrn	Stores the return integer value from the system call.
shmid	Stores and passes the shared memory segment identifier to the system call.
command	Stores the code for the desired control command so that subsequent processing can be performed on it.
choice	Determines the member of the IPC_SET control command to be changed.
shmid_ds	Receives the specified shared memory segment identifier's data structure when an IPC_STAT control command is performed.
***buf**	A pointer passed to the system call which locates the data structure in the user memory area where the IPC_STAT control command is to place its return values or where the IPC_SET command gets the values to set.

It is important to observe that although the ***buf** pointer is declared to be a pointer to a data structure of the **shmid_ds** type, it must also be initialized to contain the address of the user memory area data structure (line 17).

The sample program for the **shmctl**(2) system call is shown here:

```
1  /*This is a program to illustrate
2  *the shared memory control, shmctl(),
3  *system call capabilities.
```

```
4  */

5  /*Include necessary header files.*/
6  #include    <stdio.h>
7  #include    <sys/types.h>
8  #include    <sys/ipc.h>
9  #include    <sys/shm.h>

10  /*Start of main C language program*/
11  main()
12  {
13      extern int errno;
14      int uid, gid, mode;
15      int rtrn, shmid, command, choice;
16      struct shmid_ds shmid_ds, *buf;
17      buf = &shmid_ds;

18      /*Get the shmid, and command.*/
19      printf("Enter the shmid = ");
20      scanf("%d", &shmid);
21      printf("\nEnter the number for\n");
22      printf("the desired command:\n");
23      printf("IPC_STAT    =  1\n");
24      printf("IPC_SET     =  2\n");
25      printf("IPC_RMID    =  3\n");
26      printf("SHM_LOCK    =  4\n");
27
28
29        scanf("%d", &command);

30      /*Check the values.*/
31      printf ("\nshmid =%d, command = %d\n",
32          shmid, command);

33      switch (command)
34      {
35      case 1:     /*Use shmctl() to duplicate
36            the data structure for
37                    shmid in the shmid_ds area pointed
38                    to by buf and then print it out.*/
39          rtrn = shmctl(shmid, IPC_STAT,
40              buf);
```

```
41          printf ("\nThe USER ID = %d\n",
42              buf->shm_perm.uid);
43          printf ("The GROUP ID = %d\n",
44              buf->shm_perm.gid);
45          printf ("The creator's ID = %d\n",
46              buf->shm_perm.cuid);
47          printf ("The creator's group ID = %d\n",
48              buf->shm_perm.cgid);
49          printf ("The operation permissions = 0%o\n",
50              buf->shm_perm.mode);
51          printf ("The slot usage sequence\n");
52          printf ("number = 0%x\n",
53              buf->shm_perm.seq);
54          printf ("The key= 0%x\n",
55              buf->shm_perm.key);
56          printf ("The segment size = %d\n",
57              buf->shm_segsz);
58          printf ("The pid of last shmop = %d\n",
59              buf->shm_lpid);
60          printf ("The pid of creator = %d\n",
61              buf->shm_cpid);
62          printf ("The current # attached = %d\n",
63              buf->shm_nattch);
64          printf("The in memory # attached = %d\n",
65              buf->shm_cnattach);
66          printf("The last shmat time = %d\n",
67              buf->shm_atime);
68          printf("The last shmdt time = %d\n",
69              buf->shm_dtime);
70          printf("The last change time = %d\n",
71              buf->shm_ctime);
72          break;

                  /* Lines 73 - 87 deleted */

88      case 2:     /*Select and change the desired
89                     member(s) of the data structure.*/

90          /*Get the original data for this shmid
91                data structure first.*/
92          rtrn = shmctl(shmid, IPC_STAT, buf);
```

```
93          printf("\nEnter the number for the\n");
94          printf("member to be changed:\n");
95          printf("shm_perm.uid   = 1\n");
96          printf("shm_perm.gid   = 2\n");
97          printf("shm_perm.mode  = 3\n");
98          printf("Entry          = ");
99          scanf("%d", &choice);
100          /*Only one choice is allowed per
101            pass as an illegal entry will
102                cause repetitive failures until
103            shmid_ds is updated with
104                IPC_STAT.*/
105          switch(choice){
106          case 1:
107              printf("\nEnter USER ID = ");
108              scanf ("%d", &uid);
109              buf->shm_perm.uid = uid;
110              printf("\nUSER ID = %d\n",
111                  buf->shm_perm.uid);
112              break;

113          case 2:
114              printf("\nEnter GROUP ID = ");
115              scanf("%d", &gid);
116              buf->shm_perm.gid = gid;
117              printf("\nGROUP ID = %d\n",
118                  buf->shm_perm.gid);
119              break;

120          case 3:
121              printf("\nEnter MODE = ");
122              scanf("%o", &mode);
123              buf->shm_perm.mode = mode;
124              printf("\nMODE = 0%o\n",
125                  buf->shm_perm.mode);
126              break;
127          }
128          /*Do the change.*/
129          rtrn = shmctl(shmid, IPC_SET,
130              buf);
131          break;
132      case 3:    /*Remove the shmid along with its
```

```
133                     associated
134                     data structure.*/
135             rtrn = shmctl(shmid, IPC_RMID, NULL);
136             break;

137         }
138         /*Perform if the call is unsuccessful.*/
139         if(rtrn == -1)
140         {
141             printf ("\nThe shmctl system call failed!\n");
142             printf ("The error number = %d\n", errno);
143         }
144         /*Return the shmid upon successful completion.*/
145         else
146             printf ("\nShmctl was
.       successful for shmid = %d\n",
147                 shmid);
148         exit (0);
149     }
```

First, the program prompts for a valid shared memory segment identifier which is stored in the **shmid** variable (lines 18-20). This is required for every **shmctl**(2) system call.

Then, the code for the desired control command must be entered (lines 21-29), and it is stored in the command variable. The code is tested to determine the control command for subsequent processing.

If the IPC_STAT control command is selected (code 1), the system call is performed (lines 39, 40) and the status information returned is printed out (lines 41-71). Note that if the system call is unsuccessful (line 146), the status information of the last successful call is printed out. In addition, an error message is displayed and the **errno** variable is printed out (lines 148, 149). If the system call is successful, a message indicates this along with the shared memory segment identifier used (lines 151-154).

If the IPC_SET control command is selected (code 2), the first thing done is to get the current status information for the message queue identifier specified (lines 90-92). This is necessary because this sample program provides for changing only one member at a time, and the system call changes all of them. Also, if an invalid value happened to be stored in the user memory area for one of these members, it would cause repetitive failures for this control command until corrected. The next thing the program does is to prompt for a code corresponding to the member to be changed (lines 93-98). This code is

stored in the **choice** variable (line 99). Now, depending upon the member picked, the program prompts for the new value (lines 105-127). The value is placed at the address of the appropriate member in the user memory area data structure, and the system call is made (lines 128-130). Depending upon success or failure, the program returns the same messages as for IPC_STAT above.

If the IPC_RMID control command (code 3) is selected, the system call is performed (lines 132-135), and the **shmid** along with its associated message queue and data structure are removed from the operating system. Note that the ***buf** pointer is not required as an argument to perform this control command and its value can be zero or NULL. Depending upon the success or failure, the program returns the same messages as for the other control commands.

4.4.4 Operations for Shared Memory

This section explains how to use the **shmat**(2) and **shmdt**(2) system calls, and provides a sample program that exercises all of their capabilities.

Using shmop

The synopsis found in the **shmop**(2) entry in the *Reference Manual* is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

char *shmat (shmid, shmaddr, shmflg)
int shmid;
char *shmaddr;
int shmflg;

int shmdt (shmaddr)
char *shmaddr;
```

Attaching a Shared Memory Segment. The **shmat**(2) system call requires three arguments to be passed to it: **shmid**, **shmaddr**, and **shmflg**. It returns a character pointer value.

Upon successful completion, this value is the address in core memory where the process is attached to the shared memory segment. When unsuccessful, the value is –1. The argument definitions are as follows:

shmid Must be a valid, nonnegative, integer value and must have already been created by the **shmget**(2) system call.

shmaddr Can be zero or user-supplied when passed to the **shmat**(2) system call. If it is zero, the operating system picks an address that is *shm_gap* bytes beyond the highest in-use address in that process. The default value for *shm_gap* is 2 Mbytes and is configurable by the system manager. For more information, see the chapter on kernel configuration in the *System Management Guide*. It would be wise to let the operating system pick addresses so as to improve portability.

shmflg Passes the SHM_RND and SHM_RDONLY flags to the **shmat**() system call.

Further details are shown in the sample program for **shmop**(). If you have problems understanding this program, read "Using **shmget**" earlier in this chapter for more detail.

Detaching Shared Memory Segments. The **shmdt**(2) system call requires one argument to be passed to it and returns an integer value. Upon successful completion, zero is returned. When unsuccessful, **shmdt**(2) returns a –1.

Further details of this system call are discussed in the sample program. If you have problems understanding the logic manipulations in this program, read "Using **shmget**" earlier in this chapter for more detail.

Sample Program

The example program in this section is a menu-driven program that exercises all possible combinations of the **shmat**(2) and **shmdt**(2) system calls.

Lines 5-9 include the required header files as specified by the **shmop**(2) entry in the *Reference Manual.*

The variables declared for this program and their purposes are as follows:

flags	Stores the codes of SHM_RND or SHM_RDONLY for the **shmat**(2) system call.
addr	Stores the address of the shared memory segment for the **shmat**(2) and **shmdt**(2) system calls.
i	A loop counter for attaching and detaching.
attach	Stores the desired number of attach operations.
shmid	Stores and passes the desired shared memory segment identifier.
shmflg	Passes the value of flags to the **shmat**(2) system call.
retrn	Stores the return values from both system calls.
detach	Stores the desired number of detach operations.

Here is the sample program for the **shmop**(2) system calls:

```
1  /*This is a program to illustrate
2  *the shared memory operations, shmop(),
3  *system call capabilities.
4  */

5  /*Include necessary header files.*/
6  #include    <stdio.h>
7  #include    <sys/types.h>
8  #include    <sys/ipc.h>
9  #include    <sys/shm.h>
10  /*Start of main C language program*/
11  main()
12  {
13      extern int errno;
14      int flags, addr, i, attach;
15      int shmid, shmflg, retrn, detach;
16      /*Loop for attachments by this process.*/
```

```
17      printf("Enter the number of\n");
18      printf("attachments for this\n");
19      printf("process (1-4).\n");
20      printf("       Attachments = ");

21      scanf("%d", &attach);
22      printf("Number of attaches = %d\n", attach);
23      for(i = 1; i <= attach; i++) {
24          /*Enter the shared memory ID.*/
25          printf("\nEnter the shmid of\n");
26          printf("the shared memory segment to\n");
27          printf("be operated on = ");
28          scanf("%d", &shmid);
29          printf("\nshmid = %d\n", shmid);


30          /*Enter the value for shmaddr.*/
31          printf("\nEnter the value for\n");
32          printf("the shared memory address\n");
33          printf("in hexadecimal:\n");
34          printf("            Shmaddr = ");
35          scanf("%x", &addr);
36          printf("The desired address = 0x%x\n", addr);

37          /*Specify the desired flags.*/
38          printf("\nEnter the corresponding\n");
39          printf("number for the desired\n");
40          printf("flags:\n");
41          printf("SHM_RND                 = 1\n");
42          printf("SHM_RDONLY              = 2\n");
43          printf("SHM_RND and SHM_RDONLY = 3\n");
44          printf("            Flags       = ");
45          scanf("%d", &flags);
46          switch(flags)
47          {
48          case 1:
49              shmflg = SHM_RND;
50              break;
51          case 2:
52              shmflg = SHM_RDONLY;
53              break;
54          case 3:
```

```
55              shmflg = SHM_RND | SHM_RDONLY;
56              break;
57          }
58          printf("\nFlags = 0%o\n", shmflg);

59          /*Do the shmat system call.*/
60          retrn = (int)shmat(shmid, addr, shmflg);
61          if(retrn == -1)  {
62              printf("\nShmat failed.  ");
63              printf("Error = %d\n", errno);
64          }
65          else {
66              printf ("\nShmat was successful\n");
67              printf("for shmid = %d\n", shmid);
68              printf("The address = 0x%x\n", retrn);
69          }
70      }

71      /*Loop for detachments by this process.*/
72      printf("Enter the number of\n");
73      printf("detachments for this\n");
74      printf("process (1-4).\n");
75      printf("       Detachments = ");
76      scanf("%d", &detach);
77      printf("Number of attaches = %d\n", detach);
78      for(i = 1; i <= detach; i++) {

79          /*Enter the value for shmaddr.*/
80          printf("\nEnter the value for\n");
81          printf("the shared memory address\n");
82          printf("in hexadecimal:\n");
83          printf("             Shmaddr = ");
84          scanf("%x", &addr);
85          printf("The desired address = 0x%x\n", addr);

86          /*Do the shmdt system call.*/
87          retrn = (int)shmdt(addr);
88          if(retrn == -1)  {
89              printf("Error = %d\n", errno);
90          }
91          else {
92              printf ("\nShmdt was successful\n");
```

```
93                  printf("for address  = 0%x\n", addr);

94              }
95          }
96      }
```

This sample program combines both the **shmat**(2) and **shmdt**(2) system calls. The program prompts for the number of attachments and enters a loop until they are done for the specified shared memory identifiers. Then, the program prompts for the number of detachments to be performed and enters a loop until they are done for the specified shared memory segment addresses.

shmat. The program prompts for the number of attachments to be performed, and the value is stored in the **attach** variable (lines 17-21).

A loop is entered using the attach variable and the **i** counter (lines 23-70) to perform the specified number of attachments.

In this loop, the program prompts for a shared memory segment identifier (lines 24-27) and it is stored in the **shmid** variable (line 28). Next, the program prompts for the address where the segment is to be attached (lines 30-34), and it is stored in the **addr** variable (line 35). Then, the program prompts for the desired flags to be used for the attachment (lines 37-44), and the code representing the flags is stored in the **flags** variable (line 45). The flags variable is tested to determine the code to be stored for the **shmflg** variable used to pass them to the **shmat**(2) system call (lines 46-57). The system call is made (line 60). If successful, a message stating so is displayed along with the attach address (lines 66-68). If unsuccessful, a message stating so is displayed and the error code is displayed (lines 62, 63). The loop then continues until it finishes.

shmdt. After the attach loop completes, the program prompts for the number of detach operations to be performed (lines 71-75), and the value is stored in the **detach** variable (line 76). sp .5 A loop is entered using the **detach** variable and the **i** counter (lines 78-95) to perform the specified number of detachments.

In this loop, the program prompts for the address of the shared memory segment to be detached (lines 79-83), and it is stored in the **addr** variable (line 84). Then, the **shmdt**(2) system call is performed (line 87). If successful, a message stating so is displayed along with the address that the segment was detached from (lines 92,93). If unsuccessful, the error number is displayed (line 89). The loop continues until it finishes.

Chapter 5
curses/terminfo

5.1 Introduction 5-1
Organization of this Chapter 5-2

5.2 Overview 5-2
What is **curses**? 5-2
What is **terminfo**? 5-4
How **curses** and **terminfo** Work Together 5-5
Other Components of the Screen Management System 5-6

5.3 Working with curses Routines 5-9
What Every **curses** Program Needs 5-10
The Header File <curses.h> 5-10
The Routines **initscr**(), **refresh**(), **endwin**() 5-11
Compiling a **curses** Program 5-13
Running a **curses** Program 5-13
More about **refresh**() and Windows 5-14
Getting Simple Output and Input 5-18
Output 5-18
addch() 5-19
Input 5-28
Controlling Output and Input 5-35
Output Attributes 5-35
Color Manipulation 5-39
Bells, Whistles, and Flashing Lights 5-48
Input Options 5-49
Building Windows and Pads 5-54
Output and Input 5-54
The Routines **wnoutrefresh**() and **doupdate**() 5-55
New Windows 5-59
Using Advanced **curses** Features 5-63
Routines for Drawing Lines and Other Graphics 5-63
Routines for Using Soft Labels 5-64
Working with More than One Terminal 5-66

5.4 Working with terminfo Routines 5-68
What Every **terminfo** Program Needs 5-68

Compiling and Running a terminfo Program 5-69
A Sample terminfo Program 5-69

5.5 Working with the terminfo Database 5-73
Writing Terminal Descriptions 5-73
Name the Terminal 5-74
Learn About the Capabilities 5-74
Specifying Capabilities 5-75
Compile the Description 5-81
Test the Description 5-81
Comparing or Printing terminfo Descriptions 5-82
Converting a termcap Description to a terminfo Description 5-83

5.6 curses Program Examples 5-84
The editor Program 5-84
The highlight Program 5-91
The scatter Program 5-92
The show Program 5-94
The two Program 5-96
The window Program 5-99
The colors Program 5-101

Figures

5-1 The stdscr/terminal screen relationship 5-15
5-2 Multiple windows and pads 5-17
5-3 The window/terminal screen relationship 5-57

Tables

5-1 Components of the curses/terminfo Screen Management System 5-7
5-2 The Default Colors Table 5-40
5-3 Example of a Pairs Table 5-41
5-4 Input Option Settings for curses Programs 5-50
5-5 Building Up a Terminal's cup Sequence 5-80

Chapter 5
curses/terminfo

5.1 Introduction

Screen management programs are a common component of many commercial computer applications. These programs handle input and output at a video display terminal. A screen program might move a cursor, print a menu, divide a terminal screen into windows, change the definitions of colors, or draw a display on the screen to help users enter and retrieve information from a database.

This chapter explains how to use the **curses** library, **terminfo** database, and the **terminfo** support routines to write terminal-independent screen management programs on a your system. It also contains information on other operating system tools that support the **curses/terminfo** screen management system.

The purpose of this chapter is to explain how to write screen management programs as quickly as possible. It does not attempt to cover every detail. Use this chapter to get familiar with the way these routines work, then use the man pages for more information.

Before attempting to use **curses/terminfo**, or to understand this chapter, you should be familiar with the following items:

- The C programming language
- The operating system/C language standard I/O package **stdio**(3S)
- "System Calls and Subroutines" and "Input/Output" in Chapter 1 of this manual

Organization of this Chapter

This chapter has the following sections:

- Overview

  Briefly describes **curses**, **terminfo**, and other related support tools.

- Working with **curses** Routines

  Contains a brief description of the routines that make up the **curses**(3X) library. These routines write to a screen, read from a screen, build windows, draw line graphics, use a terminal's soft labels, manipulate colors, and work with more than one terminal at a time. Examples are included.

- Working with **terminfo** Routines

  Describes a subset of routines in the **curses** library. These routines access and manipulate data in the **terminfo** database. They are used to set up and handle special terminal capabilities such as programmable function keys.

- Working with the **terminfo** Database

  Describes the **terminfo** database, related support tools, and their relationship to the **curses** library.

- **curses** Program Examples

  Includes programs that illustrate uses of **curses** routines.

5.2 Overview

This section offers an overview of **curses** and **terminfo** and begins to explain how they work together.

5.2.1 What is curses?

curses(3X) is the library of routines that you use to write screen management programs on the operating system. The routines are C functions and macros, with an argument syntax modeled after routines in the standard C library. For example, the routine **printw**( ) uses the same arguments as **printf**(3S). The only difference is that **printw**( ) sends its output to **stdscr** (defined in **<curses.h>**) instead of **stdout**. The routine **getch**( ) and the standard **getc**(3S) are related in the same manner. The automatic teller program at your bank might use **printw**( ) to print its menus and **getch**( ) to

accept your request for a withdrawal.

The **curses** routines are usually located in */usr/lib/libcurses.a*. To compile a program using these routines, you must use the **cc**(1) command and include **-lcurses** on the command line to tell the link editor to locate, load, and link them:

cc *file*.**c** **-lcurses** **-o** *file*

The name **curses** comes from the cursor optimization that this library of routines provides. Cursor optimization minimizes the amount a cursor has to move around a screen in order to update it. For example, if you designed a screen editor program with **curses** routines and edited the sentence

```
curses/terminfo is a great package for creating screens.
```

to read

```
curses/terminfo is the best package for creating screens.
```

the program would not output the whole line, but only the part from

```
the best
```

on. The other characters would be preserved. Because the amount of data transmitted—the output—is minimized, cursor optimization is also referred to as output optimization.

Cursor optimization takes care of updating the screen in a manner appropriate for the terminal on which a **curses** program is run. This means that the **curses** library can do whatever is required to update many different terminal types. It searches the **terminfo** database (described later) to find the correct information about a terminal.

How does cursor optimization help you and those who use your programs? First, it saves you programming time when describing how you want to update a screen. Second, it saves a user's time when the screen is updated. Third, it reduces the load on your system's communication lines when the updating takes place. Fourth, you don't have to worry about the myriad of terminals on which your program might be run.

Here's a simple **curses** program. It uses the mandatory **# include <curses.h>, initscr( )**, and **endwin( )** routines, along with some of the basic **curses** routines, to move a cursor to the middle of any terminal screen and print the character string **BullsEye**. Each of the **curses** routines is described in "Working with **curses** Routines," following. LINES and COLS are variables declared in **<curses.h>**.

```
#include <curses.h>
```

```
main()
{
    initscr();

    move( LINES/2 - 1, COLS/2 - 4 );
    addstr("Bulls");
    refresh();
    addstr("Eye");
    refresh();
    endwin();
}
```

5.2.2 What is terminfo?

terminfo refers to both of the following items:

- A group of routines within the **curses** library that handles certain terminal capabilities. You can use these **terminfo** routines to write a filter or program the function keys, if your terminal has programmable keys. Shell programmers can use the command **tput**(1) to perform many of the manipulations provided by these routines.
- A database containing the descriptions of many terminals that can be used with **curses** programs. These descriptions specify the capabilities of a terminal and the way it performs various operations—for example, how many lines and columns it has and how its control characters are interpreted.

  Each terminal description in the database is a separate, compiled file. You use the source code that **terminfo**(4) describes to create these files and the command **tic**(1M) to compile them.

  The compiled files are normally located in the directories */usr/lib/terminfo/?*. These directories have single-character names, each of which is the first character in the name of a terminal. For example, an entry for the AT&T Teletype 5425 is normally located in the file */usr/lib/terminfo/a/att5425*.

Here's a simple shell script that uses the **terminfo** database.

```
#   Clear the screen and show the 0,0 position.

tput clear
tput cup 0 0        #   or tput home
```

```
echo "<- this is 0 0"

#    Show line 5, column 10.

tput cup 5 10
echo "<- this is 5 10"
```

5.2.3 How curses and terminfo Work Together

A screen management program linked with **curses** routines uses information from the **terminfo** database at run-time. The information tells **curses** what it needs to know about the terminal being used—what we'll call the *current terminal* from here on.

For example, suppose you are using an AT&T Teletype 5425 terminal to run the simple **curses** program shown in the earlier sample program. To put the **BullsEye** in the middle of the screen, the program needs to know how many lines and columns the terminal can display. The description of the AT&T Teletype 5425 in the **terminfo** database has this information. All the **curses** program needs to know before it goes looking for the information is the name of your terminal.

The **initscr**( ) routine at the beginning of a **curses** program calls the **terminfo** routine **setupterm**( ). If **setupterm**( ) is not told which terminal to set up for, it looks at the shell environment variable **TERM**, which is usually set and exported by your **.profile** when you log in (see **profile**(4)). Knowing the value of **TERM**, it then finds and opens the correct terminal description entry in the **terminfo** database. Then, when your program calls a **curses** routine, information that the routine needs concerning the terminal's capabilities is available.

Assume that the following sample lines are in a **.profile**:

```
TERM=5425
export TERM
tput init
```

There can be other statements between them, but these three statements must appear in this order. The first line stores the terminal name in the shell environment variable **TERM**. The second line exports the variable. The third line is a **terminfo** routine that initializes the current terminal. That is, it makes sure that the terminal is set up according to its description in the **terminfo** database. If you had these lines in your **.profile** and you ran a **curses** program, the program would get the information that it needs about

your terminal from the file */usr/lib/terminfo/a/att5425*.

5.2.4 Other Components of the Screen Management System

You've now had a brief look at the main components of the **curses/terminfo** method of screen management. The rest of this section will familiarize you with the other components of this system (as shown in Table 5-1).

Table 5-1
Components of the curses/terminfo Screen Management System

Component	Brief Description
terminfo database	Files found under */usr/lib/terminfo/?/**; these files contain compiled terminal descriptions. ? is the first letter of the terminal name, and * is the terminal name.
tic(1M)	**terminfo**(4) defines terminal description source files. **tic** compiles them into **terminfo** database files.
infocmp(1M)	A routine that prints and compares compiled **terminfo** description files.
captoinfo(1M)	A routine that converts old **termcap** files to **terminfo** database files.
terminfo(4)	Defines both the **terminfo** database files and the routines used to manipulate and instantiate the strings of data in those files.
tput(1)	A **terminfo** routine that causes a string from the **terminfo** database to be sent to the terminal, thus setting one or more parameters.
curses(3X)	A library of C routines that uses information in the **terminfo** database. The routines are terminal independent. They optimize cursor movement and allow for the easy programming of screen-handling code.
Other manual pages to read	**layers**(1), **stdio**(3S), **profile**(4), **scr_dump**(4), **term**(4), **term**(5)

The **terminfo** database has already been described as one of the main components of the **curses/terminfo** screen management system. The rules for creating a terminal description source file are in the man page **terminfo**(4). The source file is then compiled using **tic**. Unless you have created a shell environment variable called **TERMINFO** that indicates a different path, **tic** will place the compiled description file into the proper directory under */usr/lib/terminfo* (provided that you have permission to create or overwrite files in that directory). To use **tic** simply type:

tic *file_name*

You may use the **–v** option to get a running commentary. An integer from 1

to 10 may follow the option (no space) to set the level of verbosity. The default is 1.

The system uses the shell environment variable **TERMINFO** to find the terminal description files. Initializing a terminal will cause **TERMINFO** to be set to */usr/lib/terminfo* unless you have already set it to some other path (**$HOME/bin**, for example). The system will look for the definition of a specific terminal under **$TERMINFO/?/***, where *?* is the first letter of the terminal name, and * is the terminal name.

Once a terminal description file has been compiled, it is no longer human-readable. The routine **infocmp** translates a compiled description file back to source statements. Invoking the command without arguments will print out the description of the terminal defined by the shell environment variable **TERM**. A single argument is taken as the name of a terminal you want to see the source description for. With no options declared (or **–I**), you will see descriptions as defined in **terminfo**(4). There are options for seeing the C variable names (**–L**), the old **termcap** names (**–C**), and all output in **termcap** format (**–r**).

If two arguments are given, **infocmp** assumes they identify two descriptions you want to compare. If no options are given (or **–d**), the differences are printed. You may also ask for a list of capabilities that the two have in common (**–c**) or a list of capabilities that neither describes (**–n**). In all of the above cases, the output lists the Boolean fields first, the numeric fields second, and the strings third.

infocmp also has options to print, trace, sort, compare files in two different directories, and output a source file derived from the union of two or more compiled description files.

Early versions of the operating system used a different method of describing terminals, called **termcap**. You can convert a **termcap** file to a **terminfo** file by using **captoinfo**. If the command is invoked with no arguments, the shell environment variable **TERMCAP** is used to get the path and the shell environment variable **TERM** to get the terminal. If **TERMCAP** is null, the routine tries to convert */etc/termcap*. If a file name is given as an option, that is the file that will be converted. The output is to standard out, and may be piped. Options include a trace mode (**–v**), one field to a line output (**–1**), and changing the output width (**–w**).

One of the definitions given earlier for **terminfo** was that it is a group of routines within **curses** that allow you to manipulate the data in a terminal description file. This small library of routines is documented later in this chapter and in the **curses**(3X) man page. The command **tput**(1) will allow you to

perform many of these manipulations from the command line or in a shell script.

tput can always be given the *–Tterminaltype* option, but doesn't need it if the shell environment variable **TERM** is set. It can be given **init**, **reset**, or **longname** as special arguments. These initialize, reset, and print out the name of the terminal, respectively. Finally, you can use the name of a **terminfo**(4) terminal attribute or capability (called a *capname*) as an argument. These capabilities can fall into three categories: Boolean, numeric, and strings. If the *capname* you specify is a string, you may include, as an argument, a list of parameters to insert into coded places in the string (instantiation). In the earlier shell script example, *cup* is the *capname* of the string the computer sends to your terminal to move the cursor to a particular position. **tput** calls a routine that instantiates the 5 and 10 into the proper places in that string, and then sends the string to your terminal, thus moving the cursor to that position.

This completes the overview of the **curses/terminfo** screen management system. A more detailed description of each component is given in the next section. If you elect to skip this and go directly to the man pages, remember that the examples at the end of the chapter might still prove useful.

5.3 Working with curses Routines

This section describes the basic **curses** routines for creating interactive screen management programs. It begins by describing the routines and other program components that every **curses** program needs to work properly. Then it tells you how to compile and run a **curses** program. Finally, it describes the most frequently used **curses** routines that handle the following tasks:

- Write output to and read input from a terminal screen.
- Control the data output and input — for example, to print output in bold type or prevent it from echoing (printing back on a screen).
- Manipulate colors on alphanumeric terminals.
- Manipulate multiple screen images (windows).
- Draw simple graphics.
- Manipulate soft labels on a terminal screen.

- Send output to and accept input from more than one terminal.

To illustrate the effect of using these routines, this chapter includes simple sample programs as the routines are introduced. A group of more complex examples is located in the section "**curses** Program Examples," later in this chapter. These larger examples are more challenging in that they sometimes make use of routines not discussed here. Keep the **curses**(3X) man page handy for the larger examples.

5.3.1 What Every curses Program Needs

Every **curses** program needs to include the header file **<curses.h>** and call the routines **initscr**(), **refresh**() or similar related routines, and **endwin**().

The Header File <curses.h>

The header file **<curses.h>** defines several global variables and data structures and defines several **curses** routines as macros.

To begin, consider the variables and data structures defined. **<curses.h>** defines all the parameters used by **curses** routines. It also defines the integer variables LINES and COLS. When a **curses** program is run on a particular terminal, these variables are assigned the vertical and horizontal dimensions of the terminal screen, respectively, by the routine **initscr**(), described in a later section. The header file also defines the constants OK and ERR. The integer variables COLORS and COLOR_PAIRS are also defined in **<curses.h>**. These are assigned, respectively, the maximum number of colors and color-pairs the terminal can support. These variables are initialized by the **start_color**() routine. (See "Color Manipulation," later in this chapter.) Most **curses** routines have return values; the OK value is returned if a routine is properly completed, and the ERR value is returned if some error occurs.

NOTE

LINES and COLS are external (global) C variables that represent the size of a terminal screen. Two shell environment variables, LINES *and* COLUMNS, *may be set in a user's shell environment. A* **curses** *program uses the shell environment variables, if they exist, as the default values for the C variables. If they do not exist, the C variables are set by reading the* **terminfo** *database. To avoid confusion, for the remainder of this document the* **$** *distinguishes environment variables from the C variables.*

Now consider the macro definitions. **<curses.h>** defines many **curses** routines as macros that call other macros or **curses** routines. For instance, the simple routine **refresh**() is a macro. This line from **<curses.h>** shows that when **refresh** is called, it is expanded to call the **curses** routine **wrefresh**():

```
#define refresh() wrefresh(stdscr)
```

The latter routine in turn calls the two **curses** routines **wnoutrefresh**() and **doupdate**(). Many other routines also group two or three routines together to achieve a particular result.

CAUTION

Avoid macro expansion in **curses** *programs because it may cause problems with certain sophisticated C features, such as the use of automatic incrementing variables.*

One final point about **<curses.h>**: it automatically includes **<stdio.h>** and the **<termio.h>** tty driver interface file. Including either file again in a program is harmless but wasteful.

The Routines initscr(), refresh(), endwin()

The routines **initscr**(), **refresh**(), and **endwin**() initialize a terminal screen to an "in **curses** state," update the contents of the screen, and restore the terminal to an "out of **curses** state," respectively. Use the simple program introduced earlier to learn about each of these routines:

```
#include <curses.h>

main()
{
   initscr();    /* initialize terminal settings and <curses.h>
                    data structures and variables */

   move( LINES/2 - 1, COLS/2 - 4 );
   addstr("Bulls");
   refresh();    /* send output to (update) terminal screen */
   addstr("Eye");
   refresh();    /* send more output to terminal screen */
   endwin();     /* restore all terminal settings */
}
```

A **curses** program usually starts by calling **initscr**(). The program should call **initscr**() only once. Using the shell environment variable **$TERM** as described in "How **curses** and **terminfo** Work Together". This routine determines what terminal is being used. It then initializes all the declared data structures and other variables from **<curses.h>**. For example, **initscr**() would initialize LINES and COLS for the sample program on whatever terminal it was run. If the Teletype 5425 were used, this routine would initialize LINES to 24 and COLS to 80. Finally, this routine writes error messages to **stderr** and exits if errors occur.

During the execution of the program, output and input is handled by routines like **move**() and **addstr**() in the sample program. For example, the following line says to move the cursor to the left of the middle of the screen:

```
move( LINES/2 - 1, COLS/2 - 4 );
```

Then, the next line says to write the character string **Bulls**.

```
addstr("Bulls");
```

For example, if the Teletype 5425 were used, these routines would position the cursor and write the character string at (11,36).

NOTE

All **curses** *routines that move the cursor move it from its home position in the upper-left corner of a screen. The (LINES,COLS) coordinate at this position is (0,0), not (1,1). Notice that the vertical coordinate is given first and the horizontal second, which is the opposite of the more common (x,y) order of screen (or graph) coordinates. The –***1** *in the sample program takes the (0,0) position into account to place the cursor on the center line of the terminal screen.*

Routines like **move**() and **addstr**() do not actually change a physical terminal screen when they are called. The screen is updated only when **refresh**() is called. Before this, an internal representation of the screen called a *screen buffer* or *window* is updated. This important concept is discussed below under "More about **refresh**() and Windows."

Finally, a **curses** program ends by calling **endwin()**. This routine restores all terminal settings and positions the cursor at the lower-left corner of the screen.

5.3.2 Compiling a curses Program

You compile programs that include **curses** routines as C language programs using the **cc**(1), which invokes the C compiler (see Chapter 1 for details).

The routines are usually stored in the library */usr/lib/libcurses.a*. To direct the link editor to search this library, you must use the **–l** option with the **cc** command.

The general command line for compiling a **curses** program is as follows:

cc *file*.**c** **–lcurses** **–o** *file*

file.**c** is the name of the source program; and *file* is the executable object module.

5.3.3 Running a curses Program

curses programs count on certain information being in a user's environment to run properly. Specifically, users of a **curses** program should usually include the following three lines in their **.profile** files:

```
TERM=current terminal type
export TERM
tput init
```

For an explanation of these lines, see "How **curses** and **terminfo** Work Together," earlier in this chapter. Users of a **curses** program could also define the environment variables INES, OLUMNS, and **$TERMINFO** in their **.profile** files. However, unlike **$TERM**, these variables do not have to be defined.

If a **curses** program does not run as expected, you might want to debug it. If so, keep a few points in mind.

First, a **curses** program is interactive and should always have knowledge of where the cursor is located. It is suggested that you do not use an interactive debugger as they usually cause changes to the contents of the screen without telling **curses** about it.

Second, a **curses** program outputs to a screen buffer until **refresh**( ) or a similar routine is called. Because visual output from the program is not immediate, debugging may be confusing.

Third, you cannot set breakpoints on **curses** routines that are macros, such as **refresh**( ). You have to set the breakpoint on routines used to define these macros. For example, you have to use **wrefresh**( ) instead of **refresh**( ). See "The Header File **<curses.h>**," earlier, for more information about macros.

5.3.4 More about refresh( ) and Windows

As mentioned earlier, **curses** routines do not update the terminal screen until **refresh**( ) is called. Instead, they write to an internal representation of the screen called a screen buffer or window. When **refresh**( ) is called, all the accumulated output is sent from the window to the current terminal screen.

<curses.h> supplies a default window called "standard screen." The C name for this is **stdscr**. It is the size of the current terminal's screen. **<curses.h>** defines **stdscr** to be of the type **WINDOW***, a pointer to a C structure which is a two-dimensional array of characters representing the terminal screen. The program always keeps track of what is on the physical (terminal) screen, as well as what is in **stdscr**. When **refresh**( ) is called, it compares the two screen images and sends a stream of characters to the terminal that make the physical screen look like **stdscr**. A **curses** program considers many different ways to do this, taking into account the various capabilities of the terminal and similarities between what is on the screen and what is in the window. It optimizes output by printing as few characters as possible. Figure 5-1 illustrates what happens when you execute the sample curses program given earlier. Notice in the figure that the terminal screen retains whatever garbage is on it until the first **refresh**( ) is called.

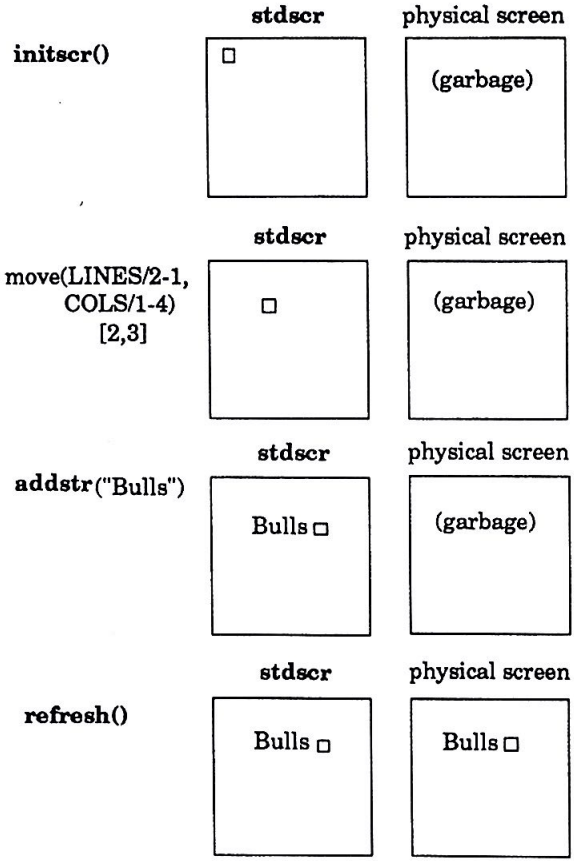


Figure 5-1. The stdscr/terminal screen relationship.

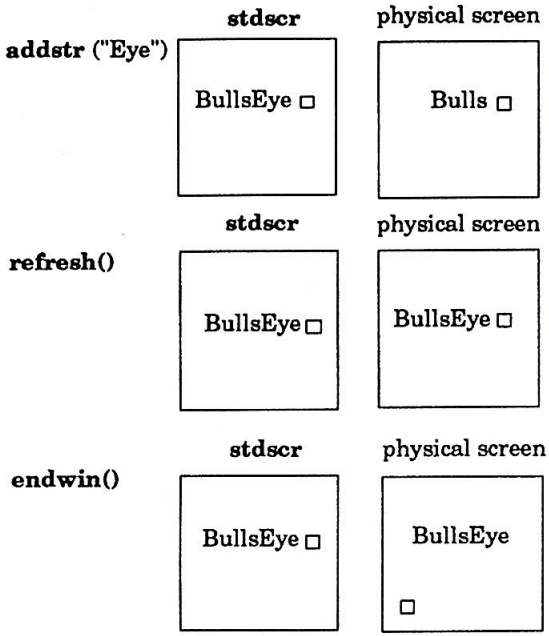


Figure 5-1. The stdscr/terminal screen relationship (cont.).

You can create other windows and use them instead of **stdscr**. Windows are useful for maintaining several different screen images. For example, many data entry and retrieval applications use two windows: one to control input and output and one to print error messages that don't affect the other window.

It's possible to divide a screen into many windows, refreshing each one of them as desired. When windows overlap, the contents of the current screen show the most recently refreshed window. It's also possible to create a window within a window. The smaller window is called a *subwindow*. Assume that you are designing an application that uses forms, (like an expense voucher), as a user interface. You could use subwindows to control access to certain fields on the form.

Some **curses** routines are designed to work with a special type of window called a *pad*. A *pad* is a window whose size is not restricted by the size of a screen or associated with a particular part of a screen. You can use a pad when you have a particularly large window or only need part of the window on the screen at any one time. For example, you might use a pad for an application with a spread sheet.

Figure 5-2 represents what a pad, a subwindow, and other windows could look like in comparison to a terminal screen.

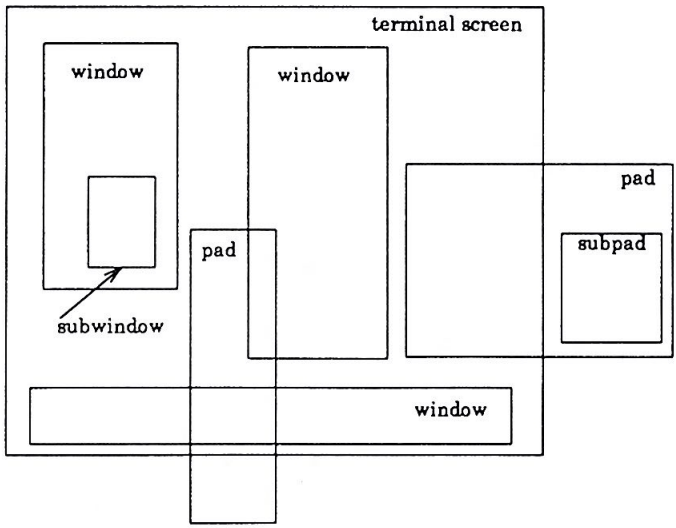


Figure 5-2. Multiple windows and pads.

The section "Building Windows and Pads" in this chapter describes the routines you use to create and use them. If you'd like to see a **curses** program with windows now, turn to the **window** program under "**curses** Program Examples," later in this chapter.

5.3.5 Getting Simple Output and Input

this section introduces how to manipulate output and input for **curses** routines.

Output

The routines that **curses** provides for writing to **stdscr** are similar to those provided by the **stdio**(3S) library for writing to a file. They let you perform the following tasks:

- Write a character at a time — **addch**()
- Write a string — **addstr**()
- Format a string from a variety of input arguments — **printw**()
- Move a cursor or move a cursor and print character(s) — **move**(), **mvaddch**(), **mvaddstr**(), **mvprintw**()
- Clear a screen or a part of it — **clear**(), **erase**(), **clrtoeol**(), **clrtobot**()

Following are descriptions and examples of these routines.

CAUTION

The **curses** *library provides its own set of output and input functions. You should not use other I/O routines or system calls, like* **read**(2) *and* **write**(2), *in a* **curses** *program. They may cause undesirable results when you run the program.*

addch()

Synopsis

```
#include <curses.h>

int addch(ch)
chtype ch;
```

Notes

- **addch()** writes a single character to **stdscr**.
- The character is of the type **chtype**, which is defined in **<curses.h>**. **chtype** contains data and attributes (see "Output Attributes" in this chapter for information about attributes).
- When working with variables of this type, make sure you declare them as **chtype** and not as the basic type (for example, **short**) that **chtype** is declared to be in **<curses.h>**. This will ensure future compatibility.
- **addch()** does some translations. For example, it makes the following conversions:
  - The **<NL>** character to a clear to end of line and a move to the next line
  - The tab character to an appropriate number of blanks
  - Other control characters to their ^*X* notation
- **addch()** normally returns **OK**. The only time **addch()** returns **ERR** is after adding a character to the lower-right corner of a window that does not scroll.
- **addch()** is a macro.

Example.

```
#include <curses.h>

main()
{
    initscr();
    addch('a');
    refresh();
    endwin();
}
```

The output from this program appears as follows:

```
a



$ □
```

Also see the **show** program under "**curses** Program Examples."

addstr( ).

Synopsis

```
#include <curses.h>

int addstr(str)
char *str;
```

Notes

- **addstr( )** writes a string of characters to **stdscr.**
- **addstr( )** calls **addch( )** to write each character.
- **addstr( )** follows the same translation rules as **addch( ).**
- **addstr( )** returns **OK** on success and **ERR** on error.
- **addstr( )** is a macro.

Example. Refer to the sample program that prints the character string **BullsEye.**

printw( ).

Synopsis

```
#include <curses.h>

int printw(fmt [, arg...] )
char * fmt
```

Notes

- **printw( )** handles formatted printing on **stdscr**.
- Like **printf**, **printw( )** takes a format string and a variable number of arguments.
- Like **addstr( )**, **printw( )** calls **addch( )** to write the string.
- **printw( )** returns **OK** on success and **ERR** on error.

Example

```
#include <curses.h>

main()
{
   char* title = "Not specified";
   int no = 0;

      /* Missing code.
*/

   initscr();

      /* Missing code.
*/

   printw("%s is not in stock.\n", title);
   printw("Please ask the cashier to order %d for you.\n", no);

   refresh();
   endwin();
}
```

The output from this program will appear as follows:

```
Not specified is not in stock.
Please ask the cashier to order 0 for you.


$ □
```

move().

Synopsis

```
#include <curses.h>

int move(y, x);
int y, x;
```

Notes

- **move**( ) positions the cursor for **stdscr** at the given row **y** and the given column **x**.
- Notice that **move**( ) takes the **y** coordinate before the **x** coordinate. The upper-left coordinates for **stdscr** are (0,0), the lower-right (LINES - 1, COLS - 1). See "The Routines **initscr**( ), **refresh**( ), and **endwin**( )" for more information.
- **move**( ) may be combined with the write functions to form the following constructs:
  - **mvaddch**(**y, x, ch**), which moves to a given position and prints a character
  - **mvaddstr**(**y, x, str**), which moves to a given position and prints a string of characters
  - **mvprintw**(**y, x, fmt** [, *arg*...]), which moves to a given position and prints a formatted string.
- **move**( ) returns **OK** on success and **ERR** on error. Trying to move to a screen position of less than (0,0) or more than (LINES - 1, COLS - 1) causes an error.
- **move**( ) is a macro.

Example

```
#include <curses.h>

main()
{
    initscr();
    addstr("Cursor should be here --> if move() works.");
    printw("\n\n\nPress <CR> to end test.");
    move(0,25);
    refresh();
    getch();      /* Gets <CR>; discussed below. */
    endwin();
}
```

Here's the output generated by running this program:

```
Cursor should be here --> □if move() works.

Press <CR> to end test.
```

After you press <CR>, the screen looks like this:

```
Cursor should be here --> □

Press <CR> to end test.
$ □
```

See the **scatter** program under **"curses** Program Examples" in this chapter for another example of using **move()**.

clear( ) and erase( ).

Synopsis

```
#include <curses.h>

int clear( )
int erase( )
```

Notes

- Both routines change **stdscr** to all blanks.
- **clear**( ) also assumes that the screen may have garbage that it doesn't know about; this routine first calls **erase**( ) and then **clearok**( ) which clears the physical screen completely on the next call to **refresh**( ) for **stdscr**. See the **curses**(3X) man page for more information about **clearok**( ).
- **initscr**( ) automatically calls **clear**( ).
- **clear**( ) always returns OK; **erase**( ) returns no useful value.
- Both routines are macros.

clrtoeol() and clrtobot().

Synopsis

#include <curses.h>

int clrtoeol()
int clrtobot()

Notes

- **clrtoeol()** changes the remainder of a line to all blanks.
- **clrtobot()** changes the remainder of a screen to all blanks.
- Both begin at the current cursor position inclusive.
- Neither returns any useful value.

Example

```
#include <curses.h>

main()
{
   initscr();
   noecho();
   addstr("Press any key to delete from here to the
   \nend of the line and on.");
   addstr("\nDelete this too.\nAnd this.");
   move(0,30);
   refresh();
   getch();
   clrtobot();
   refresh();
   endwin();
}
```

Here's the output generated by running this program:

```
Press <CR> to delete from here□ to the
end of the line and on.
Delete this too.
And this.
```

Notice the two calls to **refresh**(): one to send the full screen of text to a terminal, the other to clear from the position indicated to the bottom of a screen.

Here's what the screen looks like when you press CR:

```
Press <CR> to delete from here

$ □
```

See the **show** and **two** programs under "**curses** Program Examples" for examples of uses for **clrtoeol**().

Input

curses routines for reading from the current terminal are similar to those provided by the **stdio**(3S) library for reading from a file. They let you perform the following tasks:

- Read a character at a time — **getch**()
- Read a **<NL>**-terminated string — **getstr**()
- Parse input, converting and assigning selected data to an argument list — **scanw**()

The primary routine is **getch**(), which processes a single input character and then returns that character. This routine is like the C library routine **getchar**()(3S) except that it makes several terminal- or system-dependent options available that are not possible with **getchar**(). For example, you can use **getch**() with the **curses** routine **keypad**(), which allows a **curses** program to interpret extra keys on a terminal, such as arrow keys, function keys, and other special keys that transmit escape sequences, and treat them as just another key. See the descriptions of **getch**() and **keypad**() on the **curses**(3X) man page for more information about **keypad**(). The following pages describe and give examples of the basic routines for getting input in a screen program.

getch( ).

Synopsis

```
#include <curses.h>

int getch( )
```

Notes

- **getch**( ) reads a single character from the current terminal.
- **getch**( ) returns the value of the character or **ERR** on "end-of-file," receipt of signals, or nonblocking read with no input.
- **getch**( ) is a macro.
- See the discussions about **echo( ), noecho( ), cbreak( ), nocbreak( ), raw( ), noraw( ), halfdelay( ), nodelay( ),** and **keypad**( ) below and in **curses**(3X).

Example

```
#include <curses.h>

main()
{
    int ch;

    initscr();
    cbreak();     /* Explained later in the section */
          /*  "Input Options" */
    addstr("Press any character:  ");
    refresh();
    ch = getch();
    printw("\n\n\nThe character entered was a
    \n'%c'.\n", ch);
    refresh();
    endwin();
}
```

The output from this program follows. The first **refresh**() sends the **addstr**() character string from **stdscr** to the terminal:

```
Press any character:  □
```

Then assume that a **w** is typed at the keyboard. **getch**() accepts the character and assigns it to **ch**. Finally, the second **refresh**() is called and the screen appears as follows:

```
Press any character:  w

The character entered was a

$ □
```

For another example of **getch**(), see the **show** program under "**curses** Program Examples" in this chapter.

getstr( ).

Synopsis

```
#include <curses.h>

int getstr(str)
char *str;
```

Notes

- **getstr**( ) reads characters and stores them in a buffer until a **<CR>**, **<NL>**, or **<ENTER>** is received from **stdscr**. **getstr**( ) does not check for buffer overflow.
- The characters read and stored are in a character string.
- **getstr**( ) is a macro; it calls **getch**( ) to read each character.
- **getstr**( ) returns ERR if **getch**( ) returns ERR to it. Otherwise it returns OK.
- See the discussions about **echo**( ), **noecho**( ), **cbreak**( ), **nocbreak**( ), **raw**( ), **noraw**( ), **halfdelay**( ), **nodelay**( ), and **keypad**( ) below and in **curses**(3X).

Example

```
#include <curses.h>

main()
{
char str[256];

   initscr();
   cbreak();       /* Explained later in the */
           /* section "Input Options" */
   addstr("Enter a character string terminated by <CR>:\n\n");
   refresh()
   getstr(str);
   printw("\n\n\nThe string entered was \n'%s'\n", str);
   refresh();
   endwin();
}
```

Assume you entered the string "I enjoy learning about the barter system." The final screen (after pressing CR) appears:

```
Enter a character string terminated by <CR>:

I enjoy learning about the barter system.


The string entered was
'I enjoy learning about the barter system.'

$ □
```

scanw( ).

Synopsis

```
#include <curses.h>

int scanw(fmt [, arg...])
char *fmt;
```

Notes

- **scanw**( ) calls **getstr**( ) and parses an input line.
- Like **scanf**(3S), **scanw**( ) uses a format string to convert and assign to a variable number of arguments.
- **scanw**( ) returns the same values as **scanf**( ).
- See **scanf**(3S) for more information.

Example

```
#include <curses.h>

main()
{
   char string[100];
   float number;

   initscr();
   cbreak();          /* Explained later in the  */
   echo();            /* section "Input Options" */
   addstr("Enter a number and a string separated by a comma: ");
   refresh();
   scanw("%f,%s",&number,string);
   clear();
   printw("The string was \"%s\" and the number was %f.",string,number);
   refresh();
   endwin();
}
```

Notice the two calls to **refresh**(). The first call updates the screen with the character string passed to **addstr**(), the second with the string returned from **scanw**(). Also notice the call to **clear**(). Assume you entered the following when prompted: **2, twin**. After running this program, your terminal screen would appear as follows:

```
The string was "twin" and the number was 2.000000.

$ □
```

5.3.6 Controlling Output and Input

This section introduces how to control output and input attributes for **curses** routines.

Output Attributes

When we talked about **addch**( ), we said that it writes a single character of the type **chtype** to **stdscr**. **chtype** has two parts: a part with information about the character itself and another part with information about a set of attributes associated with the character. The attributes allow a character to be printed in reverse video, in bold, in a particular color, underlined, and so on.

stdscr always has a set of current attributes that it associates with each character as it is written. However, using the routine **attrset**( ) and related **curses** routines described below, you can change the current attributes. Below is a list of the attributes and what they mean:

- A_BLINK — blinking
- A_BOLD — extra bright or bold
- A_DIM — half bright
- A_REVERSE — reverse video
- A_STANDOUT — a terminal's best highlighting mode
- A_UNDERLINE — underlining
- A_ALTCHARSET — alternate character set (see "Drawing Lines and Other Graphics" in this chapter)
- COLOR_PAIR(*n*) — change foreground and background colors (see "Color Manipulation" in this chapter)

To use these attributes, you must pass them as arguments to **attrset**( ) and related routines; they can also be ORed with the bitwise OR (|) to **addch**( ).

NOTE

Not all terminals are capable of displaying all attributes. If a particular terminal cannot display a requested attribute, a **curses** *program attempts to find a substitute attribute. If none is possible, the attribute is ignored.*

Let's consider a use of one of these attributes. To display a word in bold, you would use the following code:

```
...
printw("A word in ");
attrset(A_BOLD);
printw("boldface");
attrset(0);
printw(" really stands out.\n");
...
refresh();
```

Attributes can be turned on one at a time, such as **attrset**(A_BOLD) in the example, or in combination. To turn on blinking bold text, for example, you would use **attrset**(A_BLINK | A_BOLD). Individual attributes can be turned on and off with the **curses** routines **attron**( ) and **attroff**( ) without affecting other attributes. **attrset(0)** turns all attributes off, including changes you may have made to foreground and background color.

Notice the attribute called A_STANDOUT. You might use it to make text attract the attention of a user. The particular hardware attribute used for standout is the most visually pleasing attribute a terminal has. Standout is typically implemented as reverse video or bold. Many programs don't really need a specific attribute, such as bold or reverse video, but instead just need to highlight some text. For such applications, the A_STANDOUT attribute is recommended. Two convenient functions, **standout**( ) and **standend**( ) can be used to turn on and off this attribute. **standend**( ), in fact, turns off all attributes.

In addition to the attributes listed above, there are three bitmasks called A_CHARTEXT, A_ATTRIBUTES, and A_COLOR. You can use these bitmasks with the **curses** function **inch**( ) and the C logical AND (**&**) operator to extract the character, attributes, or color-pair field of a position on a terminal screen. See the discussion of **inch**( ) on the **curses**(3X) man page.

Following are descriptions of **attrset( )** and the other **curses** routines that you can use to manipulate attributes.

attron( ), attrset( ), and attroff( ).

Synopsis

```
#include <curses.h>

int attron( attrs )
chtype attrs;

int attrset( attrs )
chtype attrs;

int attroff( attrs )
chtype attrs;
```

Notes

- **attron( )** turns on the requested attribute **attrs** in addition to any that are currently on. **attrs** is of the type **chtype** and is defined in **<curses.h>**.
- **attrset( )** turns on the requested attributes **attrs** instead of any that are currently turned on.
- **attroff( )** turns off the requested attributes **attrs** if they are on.
- The attributes may be combined using the bitwise OR (|).
- All return **OK**.

Example

See the **highlight** program under "**curses** Example Programs" in this chapter.

standout( ) and standend( ).

Synopsis

```
#include <curses.h>

int standout( )
int standend( )
```

Notes

- **standout**( ) turns on the preferred highlighting attribute, A_STANDOUT, for the current terminal. This routine is equivalent to **attron**(A_STANDOUT).
- **standend**( ) turns off all attributes. This routine is equivalent to **attrset**(0).
- Both always return **OK**.

Example

See the **highlight** program under "**curses** Program Examples" in this chapter.

Color Manipulation

The **curses** color manipulation routines allow you to use colors on an alphanumeric terminal as you would use any other video attribute. You can find out if the **curses** library on your system supports the color routines by checking the file */usr/include/curses.h* to see if it defines the macro **COLOR_PAIR**(*n*).

This section begins with a general description of the color feature. Then, the use of color as an attribute is explained. Next, the ways to define color-pairs and change the definitions of colors is explained. Finally, there are guidelines for ensuring the portability of your program, and a section describing the color manipulation routines and macros, with examples.

How the Color Feature Works. Colors are always used in pairs, consisting of a foreground color (used for the character) and a background color (used for the field the character is displayed on). **curses** uses this concept of color-pairs to manipulate colors. In order to use color in a **curses** program, you must first define (initialize) the individual colors, then create color-pairs using those colors, and finally, use the color-pairs as attributes.

Actually, the process is even simpler, since **curses** maintains a table of initialized colors for you. This table has as many entries as the number of colors your terminal can display at one time. Each entry in the table has three fields: one each for the intensity of the red, green, and blue components in that color.

NOTE

> **curses** *uses RGB (Red, Green, Blue) color notation. This notation allows you to specify directly the intensity of red, green, and blue light to be generated in an additive system. Some terminals use an alternative notation, known as HSL (Hue, Saturation, Luminosity) color notation. Terminals that use HSL can be identified in the* **terminfo** *database, and* **curses** *will make conversions to RGB notation automatically.*

At the beginning of any **curses** program that uses color, all entries in the colors table are initialized with eight basic colors, as shown in Table 5-2.

Table 5-2
The Default Colors Table

	Intensity of Component (R)ed	(G)reen	(B)lue
/* black: 0 */	0	0	0
/* blue: 1 */	0	0	1000
/* green: 2 */	0	1000	0
/* cyan: 3 */	0	1000	1000
/* red: 4 */	1000	0	0
/* magenta: 5 */	1000	0	1000
/* yellow: 6 */	1000	1000	0
/* white: 7 */	1000	1000	1000

Most color alphanumeric terminals can display eight colors at the same time, but if your terminal can display more than eight, then the table will have more than eight entries. The same eight colors will be used to initialize additional entries. If your terminal can display only *N* colors, where *N* is less than eight, then only the first *N* colors shown in Table 5-2 will be used.

You can change these color definitions with the routine **init_color( )** if your terminal is capable of redefining colors. (See "Changing the Definitions of Colors" for more information.)

The following color macros are defined in **curses.h** and have numeric values corresponding to their position in Table 5-2.

```
COLOR_BLACK     0
COLOR_BLUE      1
COLOR_GREEN     2
COLOR_CYAN      3
COLOR_RED       4
COLOR_MAGENTA   5
COLOR_YELLOW    6
COLOR_WHITE     7
```

curses also maintains a table of color-pairs, which has space allocated for as many entries as the number of color-pairs that can be displayed on your terminal screen at the same time. Unlike the colors table, however, there are no default entries in the pairs table: it is your responsibility to initialize any color-pair you want to use, with **init_pair( )**, before you use it as an attribute.

Each entry in the pairs table has two fields: the foreground color, and the background color. For each color-pair that you initialize, these two fields will each contain a number representing a color in the colors table. (Note that color-pairs can only be made from previously initialized colors.)

The following example pairs table shows that a programmer has used **init_pair( )** to initialize color-pair 1 as a blue foreground (entry 1 in the default color table) on yellow background (entry 6 in the default color table). Similarly, the programmer has initialized color-pair 2 as a cyan foreground on a magenta background. Noninitialized entries in the pairs table would actually contain zeros, which corresponds to black on black.

Note that color-pair 0 is reserved for use by **curses** and should not be changed or used in application programs.

Table 5-3
Example of a Pairs Table

Color-Pair Number	Foreground	Background
0 (reserved)	0	0
1	1	6
2	3	5
3	0	0
4	0	0
5	0	0
. . .	. . .	. . .

Two global variables used by the color routines are defined in **<curses.h>**. They are COLORS, which contains the maximum number of colors the terminal supports, and COLOR_PAIRS, which contains the maximum number of color-pairs the terminal supports. Both are initialized by the **start_color( )** routine to values it gets from the **terminfo** database.

Upon termination of your **curses** program, all colors and color-pairs will be restored to the values they had when the terminal was just turned on.

Using the COLOR_PAIR(*n*) Attribute. If you choose to use the default color definitions, there are only two things you need to do before you can use the attribute COLOR_PAIR(*n*). First, you must call the routine **start_color( )**. Once you've done that, you can initialize color-pairs with the routine **init_pair(*pair, f, b*)**. The first argument, *pair*, is the number of the color-pair to be initialized (or changed), and must be between **1** and COLOR_PAIRS-1. The arguments *f* and *b* are the foreground color number and the background color number. The value of these arguments must be between **0** and COLORS-1. For example, the two color-pairs in the pairs table described earlier can be initialized in the following way:

```
init_pair (1, COLOR_BLUE, COLOR_YELLOW);
init_pair (2, COLOR_CYAN, COLOR_MAGENTA);
```

Once you have initialized a color-pair, the attribute COLOR_PAIR(*n*) can be used as you would use any other attribute. COLOR_PAIR(*n*) is a macro, defined in **<curses.h>**. The argument, *n*, is the number of a previously initialized color-pair. For example, you can use the routine **attron( )** to turn on a color-pair in addition to any other attributes you may currently have turned on:

```
attron (COLOR_PAIR(1));
```

If you had initialized color-pair 1 in the way shown in Table 5-3, then characters displayed after you turned on color-pair 1 with **attron( )** would be displayed as blue characters on a yellow background.

You can also combine COLOR_PAIR(*n*) with other attributes, for example:

```
attrset(A_BLINK | COLOR_PAIR(1));
```

This would turn on blinking and whatever you have initialized color-pair 1 to be. (**attron( )** and **attrset( )** are described in "Controlling Input and Output" and also on the **curses(3X)** man page.)

Changing the Definitions of Colors. If your terminal is capable of redefining colors, you can change the predefined colors with the routine

init_color(*color, r, g, b*). The first argument, *color*, is the numeric value of the color you want to change, and the last three, *r*, *g*, and *b*, are the intensities of the red, green, and blue components, respectively, that the new color will contain. Once you change the definition of a color, all occurrences of that color on your screen change immediately.

So, for example, you could change the definition of color 1 (COLOR_BLUE by default), to be light blue, in the following way.

```
init_color (COLOR_BLUE, 0, 700, 1000);
```

If your terminal is not able to change the definition of a color, use of **init_color**( ) returns **ERR**.

Portability Guidelines. Like the rest of **curses**, the color manipulation routines have been designed to be terminal independent. But it must be remembered that the capabilities of terminals vary. For example, if you write a program for a terminal that can support 64 color-pairs, that program would not be able to produce the same color effects on a terminal that supports at most eight color-pairs.

When you are writing a program that may be used on different terminals, follow these guidelines:

- **Use a maximum of seven color-pairs made from a maximum of eight colors.**

  Programs that follow this guideline will run on most color terminals. Only seven, not eight, color-pairs should be used, even though many terminals support eight color-pairs, because **curses** reserves color-pair 0 for its own use.

- **Do not use color 0 as a background color.**

  This is recommended because on some terminals, no matter what color you have defined it to be, color 0 will always be converted to black when used for a background.

- **Combine color and other video attributes.**

  Programs that follow this guideline will provide some sort of highlighting, even if the terminal is monochrome. On color terminals, as many of the listed attributes as possible would be used. On monochrome terminals, only the video attributes would be used, and the color attribute would be ignored.

- Use the global variables COLORS and COLOR_PAIRS rather than constants when deciding how many colors or color-pairs your program should use.

Other Macros and Routines. There are two other macros defined in **<curses.h>** that you can use to obtain information from the color-pair field in characters of type **chtype**.

- A_COLOR is a bitmask to extract color-pair information. It can be used to clear the color-pair field, and to determine if any color-pair is being used.
- PAIR_NUMBER(*attrs*) is the reverse of COLOR_PAIR(n). It returns the color-pair number associated with the named attribute, *attrs*.

There are two color routines that give you information about the terminal your program is running on. The routine **has_colors( )** returns a Boolean value: TRUE if the terminal supports colors, FALSE otherwise. The routine **can_change_colors( )** also returns a Boolean value: TRUE if the terminal supports colors *and* can change their definitions, FALSE otherwise.

There are two color routines that give you information about the colors and color-pairs that are currently defined on your terminal. The routine **color_content( )** gives you a way to find the intensity of the RGB components in an initialized color. It returns ERR if the color does not exist or if the terminal cannot change color definitions, OK otherwise. The routine **pair_content( )** allows you to find out what colors a given color-pair consists of. It returns ERR is the color-pair has not been initialized, OK otherwise.

These routines are explained in more detail on the **curses**(3X) man page.

The routines **start_color( )**, **init_color( )**, and **init_pair( )** are described on the following pages, with examples of their use. You can also refer to the program **colors** in "**curses** Program Examples," at the end of this chapter, for an example of using the attribute of color in windows.

start_color( ).

Synopsis

```
#include <curses.h>

int start_color( )
```

Notes

- This routine must be called if you want to use colors, and before any other color manipulation routine is called. It is good practice to call it right after **initscr( )**.
- It initializes eight default colors (black, blue, green, cyan, red, magenta, yellow, and white), and the global variables COLORS and COLOR_PAIRS. If the value corresponding to COLOR_PAIRS in the terminfo database is greater than 64, COLOR_PAIRS will be set to 64.
- It restores the terminal's colors to the values they had when the terminal was just turned on.
- It returns ERR if the terminal does not support colors, OK otherwise.

Example

See the example under **init_pair( )**.

init_pair( ).

Synopsis

```
#include <curses.h>

int init_pair (pair, f, b)
short pair, f, b;
```

Notes

- **init_pair( )** changes the definition of a color-pair.
- Color-pairs must be initialized with **init_pair( )** before they can be used as the argument to the attribute macro COLOR_PAIR(*n*).
- The value of the first argument, *pair*, is the number of a color-pair, and must be between **1** and COLOR_PAIRS-1.
- The value of the *f* (foreground) and *b* (background) arguments must be between **0** and COLORS-1.
- If the color-pair was previously initialized, the screen will be refreshed and all occurrences of that color-pair will change to the new definition.
- It returns OK if it was able to change the definition of the color-pair, ERR otherwise.

Example

```
#include <curses.h>

main( )
{
   initscr ( );
   if (start_color ( ) == OK)
   {
      init_pair (1, COLOR_RED, COLOR_GREEN);
      attron (COLOR_PAIR (1));
      addstr ("Red on Green");
      getch( );
   }
   endwin( );
}.
```

Also see the program **colors** in "**curses** Program Examples."

init_color.

Synopsis

#include <curses.h>

int init_color(color, r, g, b)
short color, r, g, b;

Notes

- **init_color( )** changes the definition of a color.
- The first argument, *color*, is the number of the color to be changed. The value of *color* must be between **0** and COLORS-1.
- The last three arguments, *r*, *g*, and *b*, are the amounts of red, green, and blue (RBG) components in the new color. The values of these three arguments must be between **0** and **1000**.
- When **init_color( )** is used to change the definition of an entry in the colors table, all places where the old color was used on the screen immediately change to the new color.
- It returns OK if it was able to change the definition of the color, ERR otherwise.

Example

```
#include <curses.h>

main()
{
    initscr();
    if (start_color() == OK)
    {
        init_pair (1, COLOR_RED, COLOR_GREEN);
        attron (COLOR_PAIR (1));
        if (init_color (COLOR_RED, 0, 0, 1000) == OK)
            addstr ("BLUE ON GREEN");
        else
            addstr ("RED ON GREEN");
        getch ();
    }
    endwin();
}
```

Bells, Whistles, and Flashing Lights

Occasionally, you may want to get a user's attention. Two **curses** routines were designed to help you do this. They let you ring the terminal's chimes and flash its screen.

flash( ) flashes the screen if possible, and otherwise rings the bell. Flashing the screen is intended as a bell replacement, and is particularly useful if the bell bothers someone within ear shot of the user. The routine **beep**( ) can be called when a real beep is desired. (If for some reason the terminal is unable to beep, but able to flash, a call to **beep**( ) will flash the screen.)

beep( ) and flash( ).

Synopsis

#include <curses.h>

int flash( ) int beep( )

Notes

- **flash**( ) tries to flash the terminals screen, if possible; if not, tries to ring the terminal bell.
- **beep**( ) tries to ring the terminal bell, if possible: if not, tries to flash the terminal screen.
- Neither returns any useful value.

Input Options

The system does a considerable amount of processing on input before an application ever sees a character. For example, it does the following:

- Echoes (prints back) characters to a terminal as they are typed
- Interprets an erase character (typically #) and a line kill character (typically @)
- Interprets a CTRL-D (control d) as end of file (EOF)
- Interprets interrupt and quit characters
- Strips the character's parity bit
- Translates **<CR>** to **<NL>**

Because a **curses** program maintains total control over the screen, **curses** turns off echoing on the operating system and does echoing itself. At times, you may not want the system to process other characters in the standard way in an interactive screen management program. Some **curses** routines, **noecho**( ) and **cbreak**( ), for example, have been designed so that you can change the standard character processing. Using these routines in an application controls how input is interpreted. Table 5-4 shows some of the major routines for controlling input.

Every **curses** program accepting input should set some input options. This is because when the program starts running, the terminal on which it runs may be in **cbreak**( ), **raw**( ), **nocbreak**( ), or **noraw**( ) mode. Although the **curses** program starts up in **echo**( ) mode, as Table 5-4 shows, none of the other modes are guaranteed.

The combination of **noecho**( ) and **cbreak**( ) is most common in interactive screen management programs. Suppose, for instance, that you don't want the characters sent to your application program to be echoed wherever the cursor currently happens to be. Instead, you want them echoed at the bottom of the screen. The **curses** routine **noecho**( ) is designed for this purpose. However, when **noecho**( ) turns off echoing, normal erase and kill processing is still on. Using the routine **cbreak**( ) causes these characters to be uninterpreted.

Table 5-4
Input Option Settings for curses Programs

Input Options	Characters	
	Interpreted	**Uninterpreted**
Normal 'out of **curses** state'	interrupt, quit stripping **<CR>** to **<NL>** echoing erase, kill EOF	
Normal **curses** 'start up state'	echoing (simulated)	All else undefined.
cbreak() and **echo()**	interrupt, quit stripping echoing	erase, kill EOF
cbreak() and **noecho()**	interrupt, quit stripping	echoing erase, kill EOF
nocbreak() and **noecho()**	break, quit stripping erase, kill EOF	echoing
nocbreak() and **echo()**	See caution below.	
nl()	**<CR>** to **<NL>**	
nonl()		**<CR>** to **<NL>**
raw() (instead of **cbreak()**)		break, quit stripping

CAUTION

Do not use the combination **nocbreak**( ) *and* **noecho**( ). *If you use it in a program and also use* **getch**( ), *the program will go in and out of* **cbreak**( ) *mode to get each character. Depending on the state of the tty driver when each character is typed, the program may produce undesirable output.*

In addition to the routines noted in Table 5-4, you can use the **curses** routines **noraw**( ), **halfdelay**( ), and **nodelay**( ) to control input. See the **curses**(3X) man page for discussions of these routines.

The next few pages describe **noecho**( ), **cbreak**( ) and the related routines **echo**( ) and **nocbreak**( ) in more detail.

echo( ) and noecho( ).

Synopsis

#include <curses.h>

int echo( ) int noecho( )

Notes

- **echo**( ) turns on echoing of characters by **curses** as they are read in. This is the initial setting.
- **noecho**( ) turns off the echoing.
- Neither returns any useful value.
- **curses** programs may not run properly if you turn on echoing with **nocbreak**( ). See Table 5-4 and accompanying caution. After you turn echoing off, you can still echo characters with **addch**( ).

Example

See the **editor** and **show** programs under "**curses** Program Examples" in this chapter.

cbreak( ) and nocbreak( ).

Synopsis

```
#include <curses.h>
int cbreak( )
int nocbreak( )
```

Notes

- **cbreak**( ) turns on "break for each character" processing. A program gets each character as soon as it is typed, but the erase, line kill, and CTRL-D characters are not interpreted.
- **nocbreak**( ) returns to normal "line at a time" processing. This is typically the initial setting.
- Neither returns any useful value.
- A **curses** program may not run properly if **cbreak**( ) is turned on and off within the same program or if the combination **nocbreak**( ) and **echo**( ) is used.
- See Table 5-4 and accompanying caution.

Example

See the **editor** and **show** programs under "**curses** Program Examples" in this chapter.

5.3.7 Building Windows and Pads

An earlier section in this chapter, "More About **refresh**( ) and Windows" explained what windows and pads are and why you might want to use them. This section describes the **curses** routines you use to manipulate and create windows and pads.

Output and Input

The routines that you use to send output to and get input from windows and pads are similar to those you use with **stdscr**. The only difference is that you have to give the name of the window to receive the action. Generally, these functions have names formed by putting the letter **w** at the beginning of the name of a **stdscr** routine and adding the window name as the first parameter. For example, **addch**('c') would become **waddch**(**mywin, 'c'**) if you wanted to write the character **c** to the window **mywin**. Here's a list of the window (or **w**) versions of the output routines discussed in "Getting Simple Output and Input."

- **waddch**(*win, ch*)
- **mvwaddch**(*win, y, x, ch*)
- **waddstr**(*win, str*)
- **mvwaddstr**(*win, y, x, str*)
- **wprintw**(*win, fmt [, arg...]*)
- **mvwprintw**(*win, y, x, fmt [, arg...]*)
- **wmove**(*win, y, x*)
- **wclear**(*win*) and **werase**(*win*)
- **wclrtoeol**(*win*) and **wclrtobot**(*win*)
- **wrefresh**( )

You can see from their declarations that these routines differ from the versions that manipulate **stdscr** only in their names and the addition of a *win* argument. Notice that the routines whose names begin with **mvw** take the *win* argument before the *y, x* coordinates, which is contrary to what the names imply. See **curses**(3X) for more information about these routines or the versions of the input routines **getch, getstr**( ), and so on that you should use with windows.

All **w** routines can be used with pads except for **wrefresh**( ) and **wnoutrefresh**( ) (see below). In place of these two routines, you have to use **prefresh**( ) and **pnoutrefresh**( ) with pads.

The Routines wnoutrefresh( ) and doupdate( )

If you recall the earlier discussion about **refresh**( ), it sends the output from **stdscr** to the terminal screen. It was also stated that it was a macro that expands to **wrefresh(stdscr)** (see "What Every **curses** Program Needs" and "More about **refresh**( ) and Windows".

The **wrefresh**( ) routine sends the contents of a window (**stdscr** or one that you create) to a screen. It calls the routines **wnoutrefresh**( ) and **doupdate**( ). Similarly, **prefresh**( ) sends the contents of a pad to a screen by calling **pnoutrefresh**( ) and **doupdate**( ).

Using **wnoutrefresh**( )—or **pnoutrefresh**( ) (this discussion will be limited to the former routine for simplicity)—and **doupdate**( ), you can update terminal screens with more efficiency than using **wrefresh**( ) by itself. **wrefresh**( ) works by first calling **wnoutrefresh**( ), which copies the named window to a data structure referred to as the virtual screen. The virtual screen contains what a program intends to display at a terminal. After calling **wnoutrefresh**( ), **wrefresh**( ) then calls **doupdate**( ), which compares the virtual screen to the physical screen and does the actual update. If you want to output several windows at once, calling **wrefresh**( ) results in alternating calls to **wnoutrefresh**( ) and **doupdate**( ), causing several bursts of output to a screen. However, by calling **wnoutrefresh**( ) for each window and then **doupdate**( ) only once, you can minimize the total number of characters transmitted and the processor time used. The following sample program uses only one **doupdate**( ):

```
#include <curses.h>

main( )
{
   WINDOW *w1, *w2;

   initscr( );
   w1 = newwin(2,6,0,3);
   w2 = newwin(1,4,5,4);
   waddstr(w1, "Bulls");
   wnoutrefresh(w1);
   waddstr(w2, "Eye");
   wnoutrefresh(w2);
```

```
    doupdate();
    endwin();
}
```

Notice from the sample that you declare a new window at the beginning of a **curses** program. The lines

```
w1 = newwin(2,6,0,3);
w2 = newwin(1,4,5,4);
```

declare two windows named **w1** and **w2** with the routine **newwin**() according to certain specifications. **newwin**() is discussed in more detail below.

Figure 5-3 illustrates the effect of **wnoutrefresh**() and **doupdate**() on these two windows, the virtual screen, and the physical screen:

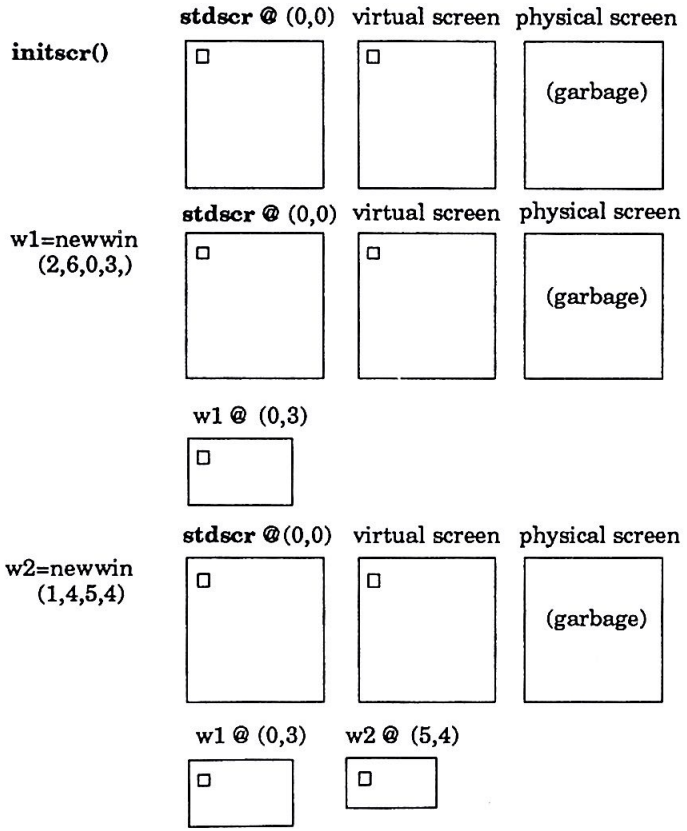


Figure 5-3. The window/terminal screen relationship.

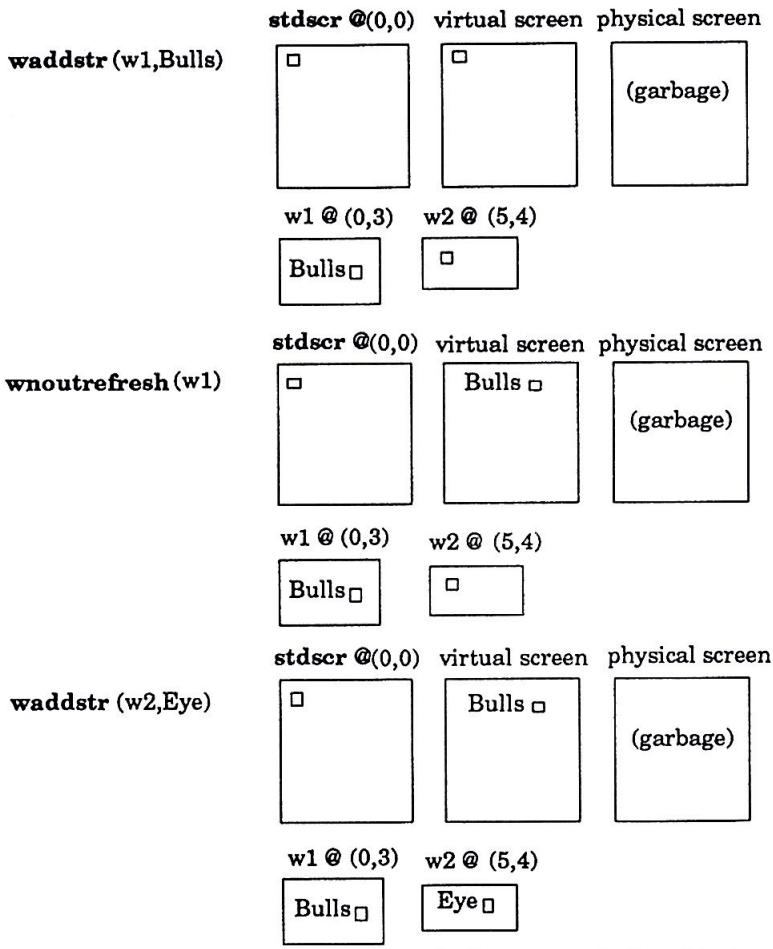


Figure 5-3. The window/terminal screen relationship (cont.).

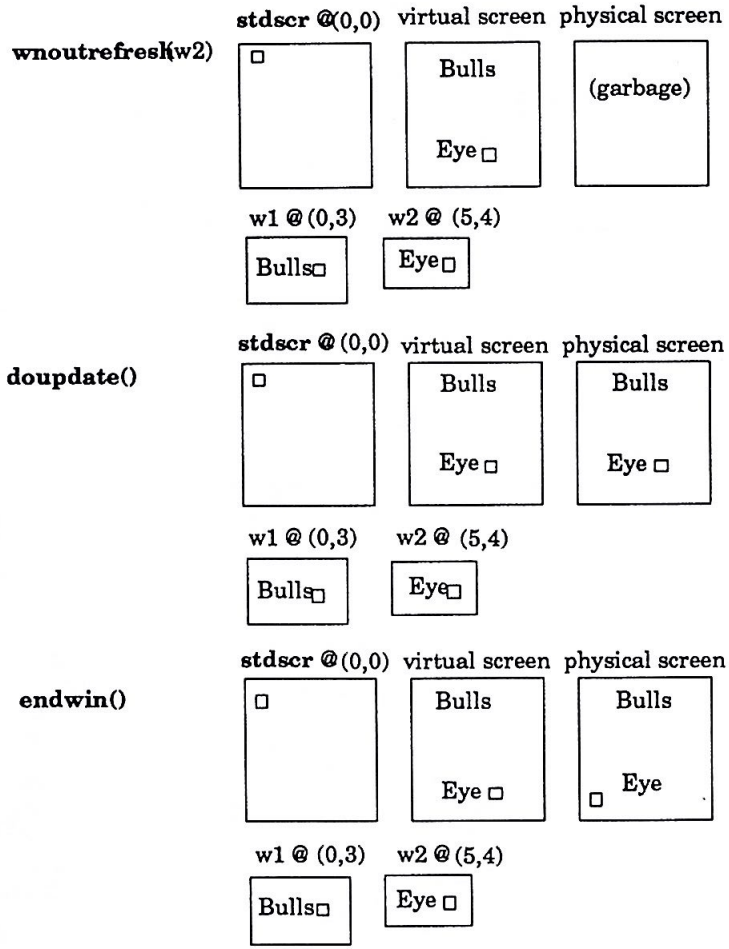


Figure 5-3. The window/terminal screen relationship (cont.).

New Windows

Following are descriptions of the routines **newwin()** and **subwin()**, which you use to create new windows. For information about creating new pads with **newpad()** and **subpad()**, see the **curses**(3X) man page.

newwin( ).

Synopsis

```
#include <curses.h>

WINDOW *newwin(nlines, ncols, begin_y, begin_x)
int nlines, ncols, begin_y, begin_x;
```

Notes

- **newwin**( ) returns a pointer to a new window with a new data area.
- The variables **nlines** and **ncols** give the size of the new window.
- **begin_y** and **begin_x** give the screen coordinates from (0,0) of the upper-left corner of the window as it is refreshed to the current screen.

Example

Recall the sample program using two windows in Figure 5-3. Also see the **window** program under "**curses** Program Examples" in this chapter.

subwin( ).

Synopsis

```
#include <curses.h>

WINDOW *subwin(orig, nlines, ncols, begin_y, begin_x)
WINDOW *orig;
int nlines, ncols, begin_y, begin_x;
```

Notes

- **subwin**( ) returns a new window that points to a section of another window, **orig**.
- **nlines** and **ncols** give the size of the new window.
- **begin_y** and **begin_x** give the screen coordinates of the upper-left corner of the window as it is refreshed to the current screen.
- Subwindows and original windows can accidentally overwrite one another.

Example

```
#include <curses.h>

main()
{
 WINDOW *sub;

  initscr();
  box(stdscr,'w','w'); /*See the curses(3X) man page for box()*/
  mvwaddstr(stdscr,7,10,"-------this is 10,10");
  mvwaddch(stdscr,8,10,'|');
  mvwaddch(stdscr,9,10,'v');
  sub = subwin(stdscr,10,20,10,10);
  box(sub,'s','s');
  wnoutrefresh(stdscr);
  wrefresh(sub);
  endwin();
}
```

This program prints a border of **w**s around the **stdscr** (the sides of your terminal screen) and a border of s's around the subwindow **sub** when it is run. For another example, see the **window** program under "**curses** Program Examples" in this chapter.

5.3.8 Using Advanced curses Features

Knowing how to use the basic **curses** routines to get output and input and to work with windows, you can design screen management programs that meet the needs of many users. The **curses** library, however, has routines that let you do more in a program than handle I/O and multiple windows. The following few pages briefly describe some of these routines and what they can help you do—namely, draw simple graphics, use a terminal's soft labels, and work with more than one terminal in a single **curses** program.

You should be comfortable using the routines previously discussed in this chapter and the other routines for I/O and window manipulation discussed on the **curses**(3X) man page before you try to use the advanced **curses** features.

CAUTION

The routines described under "Routines for Drawing Lines and Other Graphics" and "Routines for Using Soft Labels" are new features. If a program uses any of these routines, it may not run on earlier releases of the operating system.

Routines for Drawing Lines and Other Graphics

Many terminals have an alternate character set for drawing simple graphics (or glyphs or graphic symbols). You can use this character set in **curses** programs. **curses** use the same names for glyphs as the VT100 line-drawing character set.

To use the alternate character set in a **curses** program, you pass a set of variables whose names begin with ACS_ to the **curses** routine **waddch**( ) or a related routine. For example, ACS_ULCORNER is the variable for the upper-left corner glyph. If a terminal has a line-drawing character for this glyph, ACS_ULCORNER's value is the terminal's character for that glyph OR'd (|) with the bitmask A_ALTCHARSET. If no line-drawing character is available for that glyph, a standard ASCII character that approximates the glyph is stored in its place. For example, the default character for ACS_HLINE, a horizontal line, is a – (minus sign). When a close approximation is not available, a + (plus sign) is used. All the standard ACS_ names and their defaults are listed on the **curses**(3X) man page.

Part of a sample program that uses line-drawing characters follows. The example uses the **curses** routine **box**( ) to draw a box around a menu on a screen. **box**( ) uses the line drawing characters by default or when | (the pipe) and - are chosen. (See **curses**(3X).) Up and down more indicators are drawn on the box border (using ACS_UARROW and ACS_DARROW) if the menu contained within the box continues above or below the screen:

```
box(menuwin, ACS_VLINE, ACS_HLINE);
...

/* output the up/down arrows */
wmove(menuwin, maxy, maxx - 5);

/* output up arrow or horizontal line */
if (moreabove)
   waddch(menuwin, ACS_UARROW);
else
   addch(menuwin, ACS_HLINE);

/*output down arrow or horizontal line */
if (morebelow)
   waddch(menuwin, ACS_DARROW);
else
   waddch(menuwin, ACS_HLINE);
```

Here's another example. Because a default down arrow (like the lowercase letter v) isn't very discernible on a screen with many lowercase characters on it, you can change it to an uppercase V.

```
if ( ! (ACS_DARROW & A_ALTCHARSET))
   ACS_DARROW = 'V';
```

For more information, see the **curses**(3X) man page.

Routines for Using Soft Labels

Another feature available on most terminals is a set of soft labels across the bottom of the screen. A terminal's soft labels are usually matched with a set of hard function keys on the keyboard. There are usually eight of these labels, each of which is usually eight characters wide and one or two lines high.

The **curses** library has routines that provide a uniform model of eight soft labels on the screen. If a terminal does not have soft labels, the bottom line of its screen is converted into a soft label area. It is not necessary for the keyboard to have hard function keys to match the soft labels for a **curses** program to make use of them.

The next few paragraphs briefly discuss most of the **curses** routines needed to use soft labels: **slk_init**(), **slk_set**(), **slk_refresh**() and **slk_noutrefresh**(), **slk_clear**(), **slk_restore**(), **slk_attron**(), **slk_attrset**(), and **slk_attroff**().

When you use soft labels in a **curses** program, you have to call the routine **slk_int**() before **initscr**(). This sets an internal flag for **initscr**() to look at that says to use the soft labels. If **initscr**() discovers that there are fewer than eight soft labels on the screen, that they are smaller than eight characters in size, or that there is no way to program them, then it will remove a line from the bottom of **stdscr** to use for the soft labels. The size of **stdscr** and the **LINES** variable will be reduced by 1 to reflect this change. A properly written program, one that is written to use the **LINES** and **COLS** variables, will continue to run as if the line had never existed on the screen.

slk_init() takes a single argument. It determines how the labels are grouped on the screen should a line get removed from **stdscr**. The choices are between a 3-2-3 arrangement as appears on AT&T terminals, or a 4-4 arrangement as appears on Hewlett-Packard terminals. The **curses** routines adjust the width and placement of the labels to maintain the pattern. The widest label generated is eight characters.

The routine **slk_set**() takes three arguments, the label number (1-8), the string to go on the label (up to eight characters), and the justification within the label (0 = left justified, 1 = centered, and 2 = right justified).

The routine **slk_noutrefresh**() is comparable to **wnoutrefresh**() in that it copies the label information onto the internal screen image, but it does not cause the screen to be updated. Since a **wrefresh**() commonly follows, **slk_noutrefresh**() is the function that is most commonly used to output the labels.

Just as **wrefresh**() is equivalent to a **wnoutrefresh**() followed by a **doupdate**(), so too the function **slk_refresh**() is equivalent to a **slk_noutrefresh**() followed by a **doupdate**().

If **initscr**() uses the bottom line of **stdscr** to simulate soft labels, the routines **slk_attron**(), **slk_attrset**(), and **slk_attroff**() can be used to manipulate the appearance of the simulated soft labels. If you use these routines to manipulate the color of the simulated soft labels, keep in mind that soft labels are shown in reverse video by default. Note that these routines will have no effect on soft function key labels supplied by the terminal. These routines are similar to **attron**(), **attrset**(), and **attroff**() (see "Controlling Output and Input" in this chapter).

To prevent the soft labels from getting in the way of a shell escape, **slk_clear**() may be called before doing the **endwin**(). This clears the soft labels off the screen and does a **doupdate**(). The function **slk_restore**() may be used to restore them to the screen. See the **curses**(3X) man page for more information about the routines for using soft labels.

Working with More than One Terminal

A **curses** program can produce output on more than one terminal at the same time. This is useful for single-process programs that access a common database, such as multiplayer games.

Writing programs that output to multiple terminals is a difficult business, and the **curses** library does not solve all the problems you might encounter. For instance, the programs—not the library routines—must determine the file name of each terminal line, and what kind of terminal is on each of those lines. The standard method, checking $TERM in the environment, does not work, because each process can only examine its own environment.

Another problem you might face is that of multiple programs reading from one line. This situation produces a race condition and should be avoided. However, a program trying to take over another terminal cannot just shut off whatever program is currently running on that line. (Usually, security reasons would also make this inappropriate. But, for some applications, such as an interterminal communication program, or a program that takes over unused terminal lines, it would be appropriate.) A typical solution to this problem requires each user logged in on a line to run a program that notifies a master program that the user is interested in joining the master program and tells it the notification program's process ID, the name of the tty line, and the type of terminal being used. Then the program goes to sleep until the master program finishes. When done, the master program wakes up the notification program and all programs exit.

A **curses** program handles multiple terminals by always having a current terminal. All function calls always affect the current terminal. The master program should set up each terminal, saving a reference to the terminals in its own variables. When it wishes to affect a terminal, it should set the current terminal as desired, and then call ordinary **curses** routines.

References to terminals in a **curses** program have the type SCREEN*. A new terminal is initialized by calling **newterm** (*type, outfd, infd*). **newterm** returns a screen reference to the terminal being set up. *type* is a character string, naming the kind of terminal being used. *outfd* is a **stdio**(3S) file pointer (FILE*) used for output to the terminal and *infd* a file pointer for input from the terminal. This call replaces the normal call to **initscr**(), which calls **newterm(getenv("TERM"), stdout, stdin)**.

To change the current terminal, call **set_term**(*sp*) where *sp* is the screen reference to be made current. **set_term**() returns a reference to the previous terminal.

It is important to realize that each terminal has its own set of windows and options. Each terminal must be initialized separately with **newterm**(). Options such as **cbreak**() and **noecho**() must be set separately for each terminal. The functions **endwin**() and **refresh**() must be called separately for each terminal. The next example shows a typical scenario to output a message to several terminals.

```
SCREEN *term[];
...
for (i=0; i<nterm; i++)
{
    set_term(terms[i]);
    mvaddstr(0, 0, "Important message");
    refresh();
}
```

See the **two** program under "**curses** Program Examples" in this chapter for a more complete example.

5.4 Working with terminfo Routines

Some programs need to use lower-level routines (like primitives) than those offered by the **curses** routines. For such programs, the **terminfo** routines are offered. They do not manage your terminal screen, but rather give you access to strings and capabilities which you can use yourself to manipulate the terminal.

There are three circumstances when it is proper to use **terminfo** routines. The first is when you need only some screen management capabilities, (for example, making text standout on a screen). The second is when you are writing a filter. A typical filter does one transformation on an input stream without clearing the screen or addressing the cursor. If this transformation is terminal-dependent and clearing the screen is inappropriate, use of the **terminfo** routines is worthwhile. The third circumstance is when you are writing a special purpose-tool that sends a special purpose string to the terminal, such as programming a function key, setting tab stops, sending output to a printer port, or dealing with the status line. Otherwise, you are discouraged from using these routines: the higher-level **curses** routines make your program more portable to other systems and to a wider class of terminals.

NOTE

You are discouraged from using **terminfo** *routines except for the purposes noted, because* **curses** *routines take care of all the glitches present in physical terminals. When you use the* **terminfo** *routines, you must deal with the glitches yourself. Also, these routines may change and be incompatible with previous releases.*

5.4.1 What Every terminfo Program Needs

A **terminfo** program typically includes the header files and routines shown in the next example.

```
#include <curses.h>
#include <term.h>
...
   setupterm( (char*)0, 1, (int*)0 );
   ...
   putp(clear_screen);
   ...
```

```
reset_shell_mode();
exit(0);
```

The header files **<curses.h>** and **<term.h>** are required because they contain the definitions of the strings, numbers, and flags used by the **terminfo** routines. **setupterm**() takes care of initialization. Passing this routine the values **(char*)0, 1,** and **(int*)0** invokes reasonable defaults. If **setupterm**() can't figure out what kind of terminal you are on, it prints an error message and exits. **reset_shell_mode**() performs functions similar to **endwin**() and should be called before a **terminfo** program exits.

A global variable like **clear_screen** is defined by the call to **setupterm**(). It can be output using the **terminfo** routines **putp**() or **tputs**(), which gives a user more control. This string should not be directly output to the terminal using the C library routine **printf**(3S), because it contains padding information. A program that directly outputs strings will fail on terminals that require padding or that use the **xon/xoff** flow control protocol.

At the **terminfo** level, the higher-level routines like **addch**() and **getch**() are not available. It is up to you to output whatever is needed. For a list of capabilities and a description of what they do, see **terminfo**(4) (see **curses**(3X) for a list of all the **terminfo** routines).

5.4.2 Compiling and Running a terminfo Program

The general command line for compiling and the guidelines for running a program with **terminfo** routines are the same as those for compiling any other **curses** program. See "Compiling a **curses** Program" and "Running a **curses** Program" in this chapter for more information.

5.4.3 A Sample terminfo Program

The sample program **termhl** shows a simple use of **terminfo** routines. It is a version of the **highlight** program (see "**curses** Program Examples") that does not use the higher-level **curses** routines. **termhl** can be used as a filter. It includes the strings to enter bold and underline mode and to turn off all attributes.

```
/*
 * A terminfo level version of the highlight program.
 */

#include <curses.h>
#include <term.h>

int ulmode = 0;     /* Currently underlining */

main(argc, argv)
  int argc;
  char **argv;
{
   FILE *fd;
   int c, c2;
   int outch();

   if (argc > 2)
   {
      fprintf(stderr, "Usage: termhl [file]\n");
      exit(1);
   }

   if (argc == 2)
   {
      fd = fopen(argv[1], "r");
      if (fd == NULL)
      {
         perror(argv[1]);
         exit(2);
      }
   }
   else
   {
      fd = stdin;
   }
   setupterm((char*)0, 1, (int*)0);

   for (;;)
   {
      c = getc(fd);
      if (c == EOF)
```

```
            break;
            if (c == '\')
            {
               c2 = getc(fd);
               switch (c2)
               {
                  case 'B':
                    tputs(enter_bold_mode, 1, outch);
                    continue;
                  case 'U':
                    tputs(enter_underline_mode, 1, outch);
                    ulmode = 1;
                    continue;
                  case 'N':
                    tputs(exit_attribute_mode, 1, outch);
                    ulmode = 0;
                    continue;
               }
               putch(c);
               putch(c2);
            }
            else
               putch(c);
      }
      fclose(fd);
      fflush(stdout);
      resetterm();
      exit(0);
}

/*
 * This function is like putchar, but it checks for
 * underlining.
 */
putch(c)
   int c;
{
   outch(c);
   if (ulmode && underline_char)
   {
      outch('\b');
      tputs(underline_char, 1, outch);
```

```
    }
}

/*
 * Outchar is a function version of putchar
 * that can be passed to
 * tputs as a routine to call.
 */
outch(c)
    int c;
{
    putchar(c);
}
```

Let's discuss the use of the function **tputs**(*cap, affcnt, outc*) in this program to gain some insight into the **terminfo** routines. **tputs**() applies padding information. Some terminals have the capability to delay output. Their terminal descriptions in the **terminfo** database probably contain strings like **$<20>**, which means to pad for 20 milliseconds (see "Specify Capabilities" in this chapter). **tputs** generates enough pad characters to delay for the appropriate time.

tput() has three parameters. The first parameter is the string capability to be output. The second is the number of lines affected by the capability. (Some capabilities may require padding that depends on the number of lines affected. For example, **insert_line** may have to copy all lines below the current line, and may require time proportional to the number of lines copied. By convention, *affcnt* is 1 if no lines are affected. The value 1 is used, rather than 0, for safety, since *affcnt* is multiplied by the amount of time per item, and anything multiplied by 0 is 0.) The third parameter is a routine to be called with each character.

For many simple programs, *affcnt* is always 1 and *outc* always calls **putchar**. For these programs, the routine **putp**(*cap*) is a convenient abbreviation. **termhl** could be simplified by using **putp**().

Now to understand why you should use the **curses**-level routines instead of **terminfo**-level routines whenever possible, note the special check for the **underline_char** capability in this sample program. Some terminals, rather than having a code to start underlining and a code to stop underlining, have a code to underline the current character. **termhl** keeps track of the current mode, and if the current character is supposed to be underlined, outputs **underline_char**, if necessary. Low-level details such as this are precisely why the **curses** level is recommended over the **terminfo** level. **curses** takes care

of terminals with different methods of underlining and other terminal functions. Programs at the **terminfo** level must handle such details themselves.

termhl was written to illustrate a typical use of the **terminfo** routines. It is more complex than it need be in order to illustrate some properties of **terminfo** programs. The routine **vidattr** (see **curses**(3X)) could have been used instead of directly outputting **enter_bold_mode**, **enter_underline_mode**, and **exit_attribute_mode**. In fact, the program would be more robust if it did, since there are several ways to change video attribute modes.

5.5 Working with the terminfo Database

The **terminfo** database describes the many terminals with which **curses** programs, as well as some system tools like **vi**(1), can be used. Each terminal description is a compiled file containing the names that the terminal is known by and a group of comma-separated fields describing the actions and capabilities of the terminal. This section describes the **terminfo** database, related support tools, and their relationship to the **curses** library.

5.5.1 Writing Terminal Descriptions

Descriptions of many popular terminals are already described in the **terminfo** database. However, it is possible that you will want to run a **curses** program on a terminal for which there is not currently a description. In that case, you'll have to build the description.

The general procedure for building a terminal description is as follows:

1. Give the known names of the terminal.
2. Learn about, list, and define the known capabilities.
3. Compile the newly-created description entry.
4. Test the entry for correct operation.
5. Go back to step 2, add more capabilities, and repeat, as necessary.

Building a terminal description is sometimes easier when you build small parts of the description and test them as you go along. These tests can expose deficiencies in the ability to describe the terminal. Also, modifying an existing description of a similar terminal can make the building task easier.

In the next few pages, we follow each step required to build a terminal description for the fictitious terminal named "myterm."

Name the Terminal

The name of a terminal is the first information given in a **terminfo** terminal description. This string of names, assuming there is more than one name, is separated by pipe symbols (|). The first name given should be the most common abbreviation for the terminal. The last name given should be a long name that fully identifies the terminal. The long name is usually the manufacturer's formal name for the terminal. All names between the first and last entries should be known synonyms for the terminal name. All names but the formal name should be typed in lowercase letters and contain no blanks. Naturally, the formal name is entered as closely as possible to the manufacturer's name.

Here is the name string from the description of the AT&T Teletype 5420 Buffered Display Terminal:

```
5420|att5420|AT&T Teletype 5420,
```

Notice that the first name is the most commonly used abbreviation and the last is the long name. Also notice the comma at the end of the name string.

Here's the name string for our fictitious terminal, myterm:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
```

Terminal names should follow common naming conventions. These conventions start with a root name, like 5425 or myterm, for example. The root name should not contain odd characters, like hyphens, that may not be recognized as a synonym for the terminal name. Possible hardware modes or user preferences should be shown by adding a hyphen and a "mode indicator" at the end of the name. For example, the "wide mode" (which is shown by a **-w**) version of our fictitious terminal would be described as **myterm-w**. **term**(5) describes mode indicators in greater detail.

Learn About the Capabilities

After you complete the string of terminal names for your description, you have to learn about the terminal's capabilities so that you can properly describe them. To learn about the capabilities your terminal has, you should do the following:

- See the owner's manual for your terminal. It should have information about the capabilities available and the character strings that make up the sequence transmitted from the keyboard for each capability.
- Test the keys on your terminal to see what they transmit, if this information is not available in the manual. You can test the keys in one of the following ways — type:

  stty –echo; cat –vu
  Type in the keys you want to test;
  for example, see what right arrow (→) transmits.
  <CR>
  <CTRL-D>
  stty echo

  or

  cat >dev/null
  Type in the escape sequences you want to test;
  for example, see what **\E [H** transmits.
  <CTRL-D>

- The first line in each of these testing methods sets up the terminal to carry out the tests. The **<CTRL-D>** helps return the terminal to its normal settings.
- See the **terminfo**(4) man page. It lists all the capability names you have to use in a terminal description.

  The following section gives details.

Specifying Capabilities

Once you know the capabilities of your terminal, you have to describe them in your terminal description. You describe them with a string of comma-separated fields that contain the abbreviated **terminfo** name and, in some cases, the terminal's value for each capability. For example, **bel** is the abbreviated name for the beeping or ringing capability. On most terminals, a CTRL-G is the instruction that produces a beeping sound. Therefore, the beeping capability would be shown in the terminal description as **bel=^G,**.

The list of capabilities may continue onto multiple lines as long as white space (that is, tabs and spaces) begins every line but the first of the description. Comments can be included in the description by putting a # at the beginning of the line.

The **terminfo**(4) man page has a complete list of the capabilities you can use in a terminal description. This list contains the name of the capability, the abbreviated name used in the database, the two-letter code that corresponds to the old **termcap** database name, and a short description of the capability. The abbreviated name that you will use in your database descriptions is shown in the column titled "Capname."

NOTE

For a **curses** *program to run on any given terminal, its description in the* **terminfo** *database must at least include the capabilities to move a cursor in all four directions and to clear the screen.*

A terminal's character sequence (value) for a capability can be a keyed operation (like CTRL-G), a numeric value, or a parameter string containing the sequence of operations required to achieve the particular capability. In a terminal description, certain characters are used after the capability name to show what type of character sequence is required. Explanations of these characters follow:

# Indicates that a numeric value is to follow. This character follows a capability that needs a number as a value. For example, the number of columns is defined as **cols80,**.

= Indicates that the capability value is the character string that follows. This string instructs the terminal how to act and may actually be a sequence of commands. There are certain characters used in the instruction strings that have special meanings. These special characters follow:

^ This indicates that a control character is to be used. For example, the beeping sound is produced by a CTRL-G. This would be shown as **^G**.

\E or \e
These characters followed by another character indicate an escape instruction. An entry of \EC would transmit to the terminal as ESCAPE-C.

\n	These characters provide a **<NL>** character sequence.
\l	These characters provide a linefeed character sequence.
\r	These characters provide a return character sequence.
\t	These characters provide a tab character sequence.
\b	These characters provide a backspace character sequence.
\f	These characters provide a formfeed character sequence.
\s	These characters provide a space character sequence.
*****nnn*	This is a character whose octal value is *nnn*, where *nnn* can be one to three digits.
$< >	These symbols are used to show a delay in milliseconds. The desired length of delay is enclosed inside the "less than/greater than" symbols (< >). The amount of delay may be a whole number, a numeric value to one decimal place (tenths), or either form followed by an asterisk (*). The * shows that the delay will be proportional to the number of lines affected by the operation. For example, a 20-millisecond delay per line would appear as **$<20*>**. See the **terminfo**(4) man page for more information about delays and padding.

Sometimes, it may be necessary to comment out a capability so that the terminal ignores this particular field. This is done by placing a period (.) in front of the abbreviated name for the capability. For example, if you would like to comment out the beeping capability, the description entry would appear as

```
.bel=^G,
```

With this background information about specifying capabilities, let's add the capability string to our description of myterm. We'll consider basic, screen-oriented, keyboard-entered, and parameter string capabilities.

Basic Capabilities. Some capabilities common to most terminals are bells, columns, lines on the screen, and overstriking of characters, if necessary. Suppose our fictitious terminal has these and a few other capabilities, as listed below. Note that the list gives the abbreviated **terminfo** name for each capability in the parentheses following the capability description:

- An automatic wrap around to the beginning of the next line whenever the cursor reaches the right-hand margin (**am**).
- The ability to produce a beeping sound. The instruction required to produce the beeping sound is ^G (**bel**).
- An 80-column wide screen (**cols**).
- A 30-line long screen (**lines**).
- Use of xon/xoff protocol (**xon**).

By combining the name string (see "Name the Terminal") and the capability descriptions that we now have, we get the following general **terminfo** database entry:

```
myterm|mytm|mine|fancy|terminal|My FANCY terminal,
    am, bel=^G, cols#80, lines#30, xon,
```

Screen-Oriented Capabilities. Screen-oriented capabilities manipulate the contents of a screen. Our example terminal myterm has the following screen-oriented capabilities. Again, the abbreviated command associated with the given capability is shown in parentheses.

- A **<CR>** is a CTRL-M (**cr**).
- A cursor-up-one-line motion is a CTRL-K (**cuu1**).
- A cursor-down-one-line motion is a CTRL-J (**cud1**).
- Moving the cursor to the left one space is a CTRL-H (**cub1**).
- Moving the cursor to the right one space is a CTRL-L (**cuf1**).
- Entering reverse-video mode is an ESCAPE-D (**smso**).
- Exiting reverse-video mode is an ESCAPE-Z (**rmso**).
- A clear-to-the-end-of-a-line sequence is an ESCAPE-K and should have a 3-millisecond delay (**el**).
- A terminal scrolls when receiving a **<NL>** at the bottom of a page (**ind**).

The revised terminal description for myterm including these screen-oriented capabilities follows:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
     am, bel=^G, cols#80, lines#30, xon,
     cr=^M, cuu1=^K, cud1=^J, cub1=^H, cuf1=^L,
     smso=\ED, rmso=\EZ, el=\EK$<3>, ind=\n,
```

Keyboard-Entered Capabilities. Keyboard-entered capabilities are sequences generated when a key is typed on a terminal keyboard. Most terminals have, at least, a few special keys on their keyboard, such as arrow keys and the backspace key. Our sample terminal has several of these keys whose sequences are as follows:

- The backspace key generates a CTRL-H (**kbs**).
- The up-arrow key generates an ESCAPE-[A (**kcuu1**).
- The down-arrow key generates an ESCAPE-[B (**kcud1**).
- The right-arrow key generates an ESCAPE-[C (**kcuf1**).
- The left-arrow key generates an ESCAPE-[D (**kcub1**).
- The home key generates an ESCAPE-[H (**khome**).

Adding this new information to our database entry for myterm produces:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
     am, bel=^G, cols#80, lines#30, xon,
     cr=^M, cuu1=^K, cud1=^J, cub1=^H, cuf1=^L,
     smso=\ED, rmso=\EZ, el=\EK$<3>, ind=0
     kbs=^H, kcuu1=\E[A, kcud1=\E[B, kcuf1=\E[C,
     kcub1=\E[D, khome=\E[H,
```

Parameter String Capabilities. Parameter string capabilities are capabilities that can take parameters — for example, those used to position a cursor on a screen or turn on a combination of video modes. To address a cursor, the **cup** capability is used and is passed two parameters: the row and column to address. String capabilities, such as **cup** and set attributes (**sgr**) capabilities, are passed arguments in a **terminfo** program by the **tparm**( ) routine.

The arguments to string capabilities are manipulated with special '%' sequences similar to those found in a **printf**(3S) statement. In addition, many of the features found on a simple stack-based RPN calculator are available. **cup**, as noted above, takes two arguments: the row and column. **sgr** takes nine arguments, one for each of the nine video attributes. See **terminfo**(4) for the list and order of the attributes and further examples of **sgr**.

Our fancy terminal's cursor position sequence requires a row and column to be output as numbers separated by a semicolon, preceded by ESCAPE-[and followed with H. The coordinate numbers are 1-based rather than 0-based. Thus, to move to row 5, column 18, from (0,0), the sequence 'ESCAPE-[6 ; 19

H' would be output.

Integer arguments are pushed onto the stack with a **'%p'** sequence followed by the argument number, such as **'%p2'** to push the second argument. A shorthand sequence to increment the first two arguments is **'%i'**. To output the top number on the stack as a decimal, a **'%d'** sequence is used, exactly as in **printf**.

Our terminal's **cup** sequence is built up as shown in Table 5-5.

Table 5-5
Building Up a Terminal's cup Sequence

cup=	Meaning
\E[	Output ESCAPE-[
%i	Increment the two arguments
%p1	Push the 1st argument (the row) onto the stack
%d	Output the row as a decimal
;	Output a semicolon
%p2	Push the 2nd argument (the column) onto the stack
%d	Output the column as a decimal
H	Output the trailing letter

or

```
cup=\E[%i%p1%d;%p2%dH,
```

Adding this new information to our database entry for myterm produces:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
    am, bel=^G, cols#80, lines#30, xon,
    cr=^M, cuu1=^K, cud1=^J, cub1=^H, cuf1=^L,
    smso=\ED, rmso=\EZ, el=\EK$<3>, ind=0
    kbs=^H, kcuu1=\E[A, kcud1=\E[B, kcuf1=\E[C,
    kcub1=\E[D, khome=\E[H,
    cup=\E[%i%p1%d;%p2%dH,
```

See **terminfo**(4) for more information about parameter string capabilities.

Compile the Description

The **terminfo** database entries are compiled using the **tic** compiler. This compiler translates **terminfo** database entries from the source format into the compiled format.

The source file for the description is usually in a file suffixed with **.ti**. For example, the description of myterm would be in a source file named **myterm.ti**. The compiled description of myterm would usually be placed in */usr/lib/terminfo/m/myterm*, since the first letter in the description entry is **m**. Links would also be made to synonyms of **myterm** (for example, to **/f/fancy**). If the environment variable $TERMINFO were set to a directory and exported before the entry was compiled, the compiled entry would be placed in the $TERMINFO directory. All programs using the entry would then look in the new directory for the description file if $TERMINFO were set, before looking in the default */usr/lib/terminfo*. The general format for the **tic** compiler is as follows:

tic [**–v**] [**–c**] *file*

The **–v** option causes the compiler to trace its actions and output information about its progress. The **–c** option causes a check for errors; it may be combined with the **–v** option. *file* shows what file is to be compiled. If you want to compile more than one file at the same time, you have to first use **cat**(1) to join them together. The following command line shows how to compile the **terminfo** source file for our fictitious terminal:

tic –v myterm.ti<CR>
(The trace information appears as the compilation proceeds.)

Refer to the **tic**(1M) man page for more information about the compiler.

Test the Description

Let's consider three ways to test a terminal description. First, you can test it by setting the environment variable $TERMINFO to the path name of the directory containing the description. If programs run the same on the new terminal as they did on the older known terminals, then the new description is functional.

Second, you can test for correct insert-line padding by commenting out **xon** in the description and then editing (using **vi**(1)) a large file (over 100 lines) at 9600 baud and deleting about 15 lines from the middle of the screen. Type **u** (undo) several times quickly. If the terminal produces bad output, then more padding is usually required. A similar test can be used for inserting a character.

Third, you can use the **tput**(1) command. This command outputs a string or an integer according to the type of capability being described. If the capability is a Boolean expression, then **tput** sets the exit code (0 for TRUE, 1 for FALSE) and produces no output. The general format for the **tput** command is as follows:

tput [*–Ttype*] *capname*

The type of terminal you are requesting information about is identified with the *–Ttype* option. Usually, this option is not necessary because the default terminal name is taken from the environment variable $TERM. The *capname* field is used to show what capability to output from the **terminfo** database.

The following command line shows how to output the "clear screen" character sequence for the terminal being used:

tput clear
(The screen is cleared.)

The following command line shows how to output the number of columns for the terminal being used:

tput cols
(The number of columns used by the terminal appears here.)

The **tput**(1) man page contains more information on the usage and possible messages associated with this command.

5.5.2 Comparing or Printing terminfo Descriptions

Sometime you may want to compare two terminal descriptions or quickly look at a description without going to the **terminfo** source directory. The **infocmp**(1M) command was designed to help you with both of these tasks. Compare two descriptions of the same terminal; for example, the following description compares the old and new 5420 entries.

```
mkdir /tmp/old /tmp/new
TERMINFO=/tmp/old tic old5420.ti
TERMINFO=/tmp/new tic new5420.ti
infocmp -A /tmp/old -B /tmp/new -d 5420 5420
```

To print out the **terminfo** source for the 5420, type:

infocmp –I 5420

5.5.3 Converting a termcap Description to a terminfo Description

CAUTION

The **terminfo** *database is designed to take the place of the* **termcap** *database. Because of the many programs and processes that have been written with and for the* **termcap** *database, it is not feasible to do a complete cutover at one time. Any conversion from* **termcap** *to* **terminfo** *requires some experience with both databases. All entries into the databases should be handled with extreme caution. These files are important to the operation of your terminal.*

The **captoinfo**(1M) command converts **termcap**(4) descriptions to **terminfo**(4) descriptions. When a file is passed to **captoinfo**, it looks for **termcap** descriptions and writes the equivalent **terminfo** descriptions on the standard output. For example, the following line converts the file */etc/termcap* to **terminfo** source, preserving comments and other extraneous information within the file:

```
captoinfo /etc/termcap
```

The following command line looks up the current terminal in the **termcap** database, as specified by the **$TERM** and **$TERMCAP** environment variables, and converts it to **terminfo**.

```
captoinfo
```

If you must have both **termcap** and **terminfo** terminal descriptions, keep the **terminfo** description only and use **infocmp –C** to get the **termcap** descriptions. This is recommended because the **terminfo** entry will be more complete, descriptive, and accurate than the **termcap** entry possibly could be.

If you have been using cursor optimization programs with the **–ltermcap** or **–ltermlib** option in the **cc** command line, those programs will still be functional. However, these options should be replaced with the **–lcurses** option.

5.6 curses Program Examples

The following examples demonstrate uses of **curses** routines.

5.6.1 The editor Program

This program illustrates how to use **curses** routines to write a screen editor. For simplicity, **editor** keeps the buffer in **stdscr**; obviously, a real screen editor would have a separate data structure for the buffer. This program has many other simplifications: no provision is made for files of any length other than the size of the screen, for lines longer than the width of the screen, or for control characters in the file.

Several points about this program are worth making. First, it uses the **move**(), **mvaddstr**(), **flash**(), **wnoutrefresh**(), and **clrtoeol**() routines. These routines are all discussed in this chapter under "Working with **curses** Routines."

Second, the program also uses some **curses** routines that we have not discussed. For example, the function to write out a file uses the **mvinch**() routine, which returns a character in a window at a given position. The data structure used to write out a file does not keep track of the number of characters in a line or the number of lines in the file, so trailing blanks are eliminated when the file is written. The program also uses the **insch**(), **delch**(), **insertln**(), and **deleteln**() routines. These functions insert and delete a character or line. See **curses**(3X) for more information about these routines.

Third, the editor command interpreter accepts special keys, as well as ASCII characters. On one hand, new users find an editor that handles special keys easier to learn about. For example, it's easier for new users to use the arrow keys to move a cursor than it is to memorize that the letter *h* means left, *j* means down, *k* means up, and *l* means right. On the other hand, experienced users usually like having the ASCII characters to avoid moving their hands from the home row position to use special keys.

NOTE

Because not all terminals have arrow keys, your **curses** *programs will work on more terminals if there is an ASCII character associated with each special key.*

Fourth, the CTRL-L command illustrates a feature most programs using **curses** routines should have. Often, some program beyond the control of the routines writes something to the screen (for instance, a broadcast message) or some line noise affects the screen so much that the routines cannot keep track of it. A user invoking **editor** can type CTRL-L, causing the screen to be cleared and redrawn with a call to **wrefresh(curscr)**.

Finally, the input command is terminated by CTRL-D, not the escape key. It is very tempting to use escape as a command, since escape is one of the few special keys available on every keyboard. (Return and break are the only others.) However, using escape as a separate key introduces an ambiguity. Most terminals use sequences of characters beginning with escape (for example, escape sequences) to control the terminal and have special keys that send escape sequences to the computer. If a computer receives an escape from a terminal, it cannot tell whether the user depressed the escape key or whether a special key was pressed.

editor and other **curses** programs handle the ambiguity by setting a timer. If another character is received during this time, and if that character might be the beginning of a special key, the program reads more input until either a full special key is read, the time out is reached, or a character is received that could not have been generated by a special key. While this strategy works most of the time, it is not foolproof. It is possible for the user to press escape, then to type another key quickly, which causes the **curses** program to think a special key has been pressed. Also, a pause occurs until the escape can be passed to the user program, resulting in a slower response to the escape key.

Many existing programs use escape as a fundamental command, which cannot be changed without infuriating a large class of users. These programs cannot make use of special keys without dealing with this ambiguity, and at best must resort to a timeout solution. So, when designing your **curses** programs, avoid the escape key.

```
/* editor: A screen-oriented editor.  The user
 * interface is similar to a subset of vi.
 * The buffer is kept in stdscr to simplify
```

```
   * the program.
   */

#include <curses.h>

#define CTRL(c) ((c) & 037)

main(argc, argv)
int argc;
char **argv;
{
        extern void perror(), exit();
        int i, n, l;
        int c;
        int line = 0;
        FILE *fd;

        if (argc != 2)
        {
                fprintf(stderr, "Usage: %s file\n", argv[0]);
                exit(1);
        }

        fd = fopen(argv[1], "r");
        if (fd == NULL)
        {
                perror(argv[1]);
                exit(2);
        }

        initscr();
        cbreak();
        nonl();
        noecho();
        idlok(stdscr, TRUE);
        keypad(stdscr, TRUE);

        /* Read in the file */
        while ((c = getc(fd)) != EOF)
        {
                if (c == '\n')
                        line++;
```

```
            if (line > LINES - 2)
                  break;
            addch(c);
      }
      fclose(fd);

      move(0,0);
      refresh();
      edit();

      /* Write out the file */
      fd = fopen(argv[1], "w");
      for (l = 0; l < LINES - 1; l++)
      {
            n = len(l);
            for (i = 0; i < n; i++)
                  putc(mvinch(l, i) & A_CHARTEXT, fd);
            putc('\n', fd);
      }
      fclose(fd);

      endwin();
      exit(0);
}

len(lineno)
int lineno;
{
      int linelen = COLS - 1;

      while (linelen >= 0 && mvinch(lineno, linelen) == ' ')
            linelen--;
      return linelen + 1;
}

/* Global value of current cursor position */
int row, col;

edit()
{
      int c;
```

```
for (;;)
{
        move(row, col);
        refresh();
        c = getch();

        /* Editor commands */
        switch (c)
        {

        /* hjkl and arrow keys: move cursor
         * in direction indicated */
        case 'h':
        case KEY_LEFT:
                if (col > 0)
                        col--;
                else
                        flash();
                break;

        case 'j':
        case KEY_DOWN:
                if (row < LINES - 1)
                        row++;
                else
                        flash();
                break;

        case 'k':
        case KEY_UP:
                if (row > 0)
                        row--;
                else
                        flash();
                break;

        case 'l':
        case KEY_RIGHT:
                if (col < COLS - 1)
                        col++;
                else
                        flash();
```

```
        break;
/* i: enter input mode */
case KEY_IC:
case 'i':
        input();
        break;

/* x: delete current character */
case KEY_DC:
case 'x':
        delch();
        break;

/* o: open up a new line and enter input mode */
case KEY_IL:
case 'o':
        move(++row, col = 0);
        insertln();
        input();
        break;

/* d: delete current line */
case KEY_DL:
case 'd':
        deleteln();
        break;

/* ^L: redraw screen */
case KEY_CLEAR:
case CTRL('L'):
        wrefresh(curscr);
        break;

/* w: write and quit */
case 'w':
        return;
/* q: quit without writing */
case 'q':
        endwin();
        exit(2);
default:
        flash();
```

```
                    break;
                }
        }
}

/*
 * Insert mode: accept characters and insert them.
 *  End with ^D or EIC
 */
input()
{
        int c;

        standout();
        mvaddstr(LINES - 1, COLS - 20, "INPUT MODE");
        standend();
        move(row, col);
        refresh();
        for (;;)
        {
                c = getch();
                if (c == CTRL('D') || c == KEY_EIC)
                        break;
                insch(c);
                move(row, ++col);
                refresh();
        }
        move(LINES - 1, COLS - 20);
        clrtoeol();
        move(row, col);
        refresh();
}
```

5.6.2 The highlight Program

This program illustrates a use of the routine **attrset**(). **highlight** reads a text file and uses embedded escape sequences to control attributes. **\U** turns on underlining, **\B** turns on bold, and **\N** restores the default output attributes.

Note the first call to **scrollok**(), a routine that we have not previously discussed (see **curses**(3X)). This routine allows the terminal to scroll if the file is longer than one screen. When an attempt is made to draw past the bottom of the screen, **scrollok**() automatically scrolls the terminal up a line and calls **refresh**().

```
/*
 * highlight: a program to turn \U, \B, and
 * \N sequences into highlighted
 * output, allowing words to be
 * displayed underlined or in bold.
 */

#include <curses.h>

main(argc, argv)
int argc;
char **argv;
{
        FILE *fd;
        int c, c2;
        void exit(), perror();

        if (argc != 2)
        {
                fprintf(stderr, "Usage: highlight file\n");
                exit(1);
        }

        fd = fopen(argv[1], "r");

        if (fd == NULL)
        {
                perror(argv[1]);
                exit(2);
        }
```

```
        initscr();
        scrollok(stdscr, TRUE);
        nonl();
        while ((c = getc(fd)) != EOF)
        {
              if (c == '\\')
              {
                    c2 = getc(fd);
                    switch (c2)
                    {
                    case 'B':
                          attrset(A_BOLD);
                          continue;
                    case 'U':
                          attrset(A_UNDERLINE);
                          continue;
                    case 'N':
                          attrset(0);
                          continue;
                    }
                    addch(c);
                    addch(c2);
              }
              else
                    addch(c);
        }
        fclose(fd);
        refresh();
        endwin();
        exit(0);
}
```

5.6.3 The scatter Program

This program takes the first **LINES – 1** lines of characters from the standard input and displays the characters on a terminal screen in a random order. For this program to work properly, the input file should not contain tabs or nonprinting characters.

```
/*
 *     The scatter program.
 */
```

```
#include    <curses.h>
#include    <sys/types.h>

extern time_t time();

#define MAXLINES 120
#define MAXCOLS  160
char s[MAXLINES][MAXCOLS];/* Screen Array */
int  T[MAXLINES][MAXCOLS];/* Tag Array - Keeps track of *
                          * the number of characters    *
                          * printed and their positions.
*/

main()
{
      register int row = 0,col = 0;
      register int c;
      int char_count = 0;
      time_t t;
      void exit(), srand();

      initscr();
      for(row = 0;row < MAXLINES;row++)
            for(col = 0;col < MAXCOLS;col++)
                  s[row][col]=' ';

      col = row = 0;
      /* Read screen in */
      while ((c=getchar()) != EOF && row < LINES ) {

            if(c != '\n')
            {
                  /* Place char in screen array */
                  s[row][col++] = c;
                  if(c != ' ')
                        char_count++;
            }
            else
            {
                  col = 0;
                  row++;
            }
```

```
        }

        time(&t);    /* Seed the random number generator */
        srand((unsigned)t);

        while (char_count)
        {
              row = rand() % LINES;
              col = (rand() >> 2) % COLS;
              if (T[row][col] != 1 && s[row][col] != ' ')
              {
                    move(row, col);
                    addch(s[row][col]);
                    T[row][col] = 1;
                    char_count--;
                    refresh();
              }
        }
        endwin();
        exit(0);
}
```

5.6.4 The show Program

show pages through a file, showing one screen of its contents each time you depress the space bar. The program calls **cbreak**( ) so that you can depress the space bar without having to hit return; it calls **noecho**( ) to prevent the space from echoing on the screen. The **nonl**( ) routine, which we have not previously discussed, is called to enable more cursor optimization. The **idlok**( ) routine, which we also have not discussed, is called to allow insert and delete line. (See **curses**(3X) for more information about these routines). Also notice that **clrtoeol**( ) and **clrtobot**( ) are called.

By creating an input file for **show** made up of screen-sized (about 24 lines) pages, each varying slightly from the previous page, nearly any exercise for a **curses**( ) program can be created. This type of input file is called a show script.

```
#include <curses.h>
#include <signal.h>

main(argc, argv)
int argc;
```

```
char *argv[];
{
        FILE *fd;
        char linebuf[BUFSIZ];
        int line;
        void done(), perror(), exit();

        if (argc != 2)
        {
                fprintf(stderr, "usage: %s file\n", argv[0]);
                exit(1);
        }

        if ((fd=fopen(argv[1], "r")) == NULL)
        {

                perror(argv[1]);
                exit(2);
        }
        signal(SIGINT, done);

        initscr();
        noecho();
        cbreak();
        nonl();
        idlok(stdscr, TRUE);

        while(1)
        {
                move(0,0);
                for (line = 0; line < LINES; line++)
                {
                        if (!fgets(linebuf, sizeof linebuf, fd))
                        {
                                clrtobot();
                                done();
                        }
                        move(line, 0);
                        printw("%s", linebuf);
                }
                refresh();
                if (getch() == 'q')
```

```
                    done();
        }
}

void done()
{
        move(LINES - 1, 0);
        clrtoeol();
        refresh();
        endwin();
        exit(0);
}
```

5.6.5 The two Program

This program pages through a file, writing one page to the terminal from which the program is invoked and the next page to the terminal named on the command line. It then waits for a space to be typed on either terminal and writes the next page to the terminal at which the space is typed.

two is just a simple example of a two-terminal **curses** program. It does not handle notification; instead, it requires the name and type of the second terminal on the command line. As written, the command "**sleep 100000**' must be typed at the second terminal to put it to sleep while the program runs, and the user of the first terminal must have both read and write permission on the second terminal.

```
#include <curses.h>
#include <signal.h>

SCREEN *me, *you;
SCREEN *set_term();

FILE *fd, *fdyou;
char linebuf[512];

main(argc, argv)
int argc;
char **argv;
{
    void done(), exit();
    unsigned sleep();
```

```
char *getenv();
int c;

if (argc != 4)
{
  fprintf(stderr,
   "Usage: two othertty otherttytype inputfile\n");
  exit(1);
}

fd = fopen(argv[3], "r");
fdyou = fopen(argv[1], "w+");
signal(SIGINT, done);   /* die gracefully */

       /*initialize my tty*/
me = newterm(getenv("TERM"), stdout, stdin);
       /*Initialize the other terminal*/
you = newterm(argv[2], fdyou, fdyou);

set_term(me); /* Set modes for my terminal */
noecho(); /* turn off tty echo */
cbreak(); /* enter cbreak mode */
nonl();       /* Allow linefeed */
nodelay(stdscr, TRUE);  /* No hang on input */

set_term(you);    /* Set modes for other terminal */
noecho();
cbreak();
nonl();
nodelay(stdscr,TRUE);

/* Dump first screen full on my terminal */
dump_page(me);

/* Dump second screen full on the other terminal */
dump_page(you);

for (;;)  /* for each screen full */
{
     set_term(me);
     c = getch();
     if (c == 'q')   /* wait for user to read it */
```

```
        done();
        if (c == ' ')
        dump_page(me);

        set_term(you);
        c = getch();
        if (c == 'q')   /* wait for user to read it */
        done();
        if (c == ' ')
        dump_page(you);
        sleep(1);
    }
}
dump_page(term)
  SCREEN *term;
{
    int line;

    set_term(term);
    move(0, 0);
    for (line = 0; line < LINES - 1; line++) {
        if (fgets(linebuf, sizeof linebuf, fd) == NULL) {
        clrtobot();
        done();
        }
        mvaddstr(line, 0, linebuf);
    }
    standout();
    mvprintw(LINES - 1, 0, "--More--");
    standend();
    refresh();    /* sync screen */
}
/*
 * Clean up and exit.
 */
void done()
{
    /* Clean up first terminal */
    set_term(you);
    move(LINES - 1,0);   /* to lower left corner */

    clrtoeol();   /* clear bottom line */
```

```
    refresh();    /* flush out everything */
    endwin(); /* curses cleanup */

    /* Clean up second terminal */
    set_term(me);
    move(LINES - 1,0);   /* to lower left corner */
    clrtoeol();   /* clear bottom line */
    refresh();    /* flush out everything */
    endwin(); /* curses cleanup */
    exit(0);
}
```

5.6.6 The window Program

This sample program demonstrates the use of multiple windows. The main display is kept in **stdscr**. When you want to put something other than what is in **stdscr** on the physical terminal screen temporarily, a new window is created covering part of the screen. A call to **wrefresh**() for that window causes it to be written over the **stdscr** image on the terminal screen. Calling **refresh**() on **stdscr** results in the original window being redrawn on the screen. Note the calls to the **touchwin**() routine (which we have not discussed — see **curses**(3X)) that occur before writing out a window over an existing window on the terminal screen. This routine prevents screen optimization in a **curses** program. If you have trouble refreshing a new window that overlaps an old window, it may be necessary to call **touchwin**() for the new window to get it completely written out.

```
#include <curses.h>

WINDOW *cmdwin;

main()

{

        int i, c;
        char buf[120];
        void exit();

        initscr();
        nonl();
        noecho();
        cbreak();
```

```
        cmdwin = newwin(3, COLS, 0, 0);/* top 3 lines */
        for (i = 0; i < LINES; i++)
              mvprintw(i, 0, "This is line %d of stdscr", i);

        for (;;)

        {
              refresh();
              c = getch();
              switch (c)

              {

              case 'c':   /* Enter command from keyboard */
                    werase(cmdwin);
                    wprintw(cmdwin, "Enter command:");
                    wmove(cmdwin, 2, 0);
                    for (i = 0; i < COLS; i++)
                          waddch(cmdwin, '-');
                    wmove(cmdwin, 1, 0);
                    touchwin(cmdwin);
                    wrefresh(cmdwin);
                    wgetstr(cmdwin, buf);
                    touchwin(stdscr);

                    /*
                     * The command is now in buf.
                     * It should be processed here.
                     */

              case 'q':
                    endwin();
                    exit(0);
              }

        }

}
```

5.6.7 The colors Program

This program creates two windows. All characters displayed in the first window will be in red, on a blue background. All characters displayed in the second window will be in yellow, on a magenta background.

```
#include <curses.h>
#define   PAIR1  1
#define   PAIR2  2
main()
{
    WINDOW *win1, *win2;
    initscr();
    if ((start_color()) == OK)
    {
        /* create windows */
        win1 = newwin (5, 40, 0, 0);
        win2 = newwin (5, 40, 15, 40);
        /* create two color pairs */
        init_pair (PAIR1, COLOR_RED, COLOR_BLUE);
        init_pair (PAIR2, COLOR_YELLOW, COLOR_MAGENTA);
        /* turn on color attributes for each window */
        wattron (win1, COLOR_PAIR (PAIR1));
        wattron (win2, COLOR_PAIR (PAIR2));
        /* print some text in each window and exit */
        waddstr (win1, "This should be red on blue");
        waddstr (win2, "This should be yellow on magenta");
        wnoutrefresh (win1);
        wnoutrefresh (win2);
        doupdate();
        /* wait for any key before terminating */
        wgetch (win2);
    }
    endwin();
}
```

Chapter 6
Common Object File Format (COFF)

6.1 Introduction 6-1

6.2 Definitions and Conventions 6-3
Sections 6-3
Physical and Virtual Addresses 6-3
Target Machine 6-4

6.3 File Header 6-4
Magic Numbers 6-5
Flags 6-5
File Header Declaration 6-5

6.4 Optional Header Information 6-6
The Standard System a.out Header 6-6
Optional Header Declaration 6-8

6.5 Section Headers 6-8
Flags 6-9
Section Header Declaration 6-11
.bss Section Header 6-11

6.6 Sections 6-12

6.7 Relocation Information 6-12
Relocation Entry Declaration 6-13

6.8 Line Numbers 6-14
Line Number Declaration 6-15

6.9 Symbol Table 6-15
Special Symbols 6-17
Inner Blocks 6-18
Symbols and Functions 6-20
Symbol Table Entries 6-20
Symbol Names 6-21
Storage Classes 6-22
Storage Classes for Special Symbols 6-24
Symbol Value Field 6-26
Section Number Field 6-27

Section Numbers and Storage Classes 6-28
Type Entry 6-29
Type Entries and Storage Classes 6-31
Structure for Symbol Table Entries 6-32
Auxiliary Table Entries 6-33
Filenames 6-34
Sections 6-34
Tag Names 6-35
End of Structures 6-35
Functions 6-36
Arrays 6-36
End of Blocks and Functions 6-37
Beginning of Blocks and Functions 6-37
Auxiliary Entry Declaration 6-38

6.10 String Table 6-40

6.11 Access Routines 6-41

Figures

6-1 Object File Format 6-2
6-2 COFF Symbol Table 6-16
6-3 Special Symbols (**.bb** and **.eb**) 6-18
6-4 Example of the Symbol Table 6-19
6-5 Symbols for Functions 6-20
6-5 String Table 6-40

Tables

6-1 File Header Contents 6-4
6-2 File Header Flags 6-5
6-3 Optional Header Contents 6-7
6-4 System Magic Numbers 6-7
6-5 Section Header Contents 6-9
6-6 Section Header Flags 6-10
6-7 Relocation Section Contents 6-12
6-8 Relocation Types 6-13
6-9 Line Number Grouping 6-14
6-10 Special Symbols in the Symbol Table 6-17
6-11 Symbol Table Entry Format 6-21
6-12 Name Field 6-22

6-13 Storage Classes 6-23
6-14 Storage Class by Special Symbols 6-24
6-15 Restricted Storage Classes 6-25
6-16 Storage Class and Value 6-26
6-17 Section Number 6-27
6-18 Section Number and Storage Class 6-28
6-19 Fundamental Types 6-29
6-20 Derived Types 6-30
6-21 Type Entries by Storage Class 6-31
6-22 Auxiliary Symbol Table Entries 6-33
6-23 Format for Auxiliary Table Entries for Sections 6-34
6-24 Tag Names Table Entries 6-35
6-25 Table Entries for End of Structures 6-35
6-26 Table Entries for Functions 6-36
6-27 Table Entries for Arrays 6-36
6-28 End of Block and Function Entries 6-37
6-29 Format for Beginning of Block and Function 6-37
6-30 Entries for Structures, Unions, and Enumerations 6-38

Chapter 6
Common Object File Format (COFF)

6.1 Introduction

This section describes the Common Object File Format (COFF) used on Sequent computers. COFF is the format of the output file produced by the assembler, **as**, and the link editor, **ld**.

Here are some key features of COFF:

- Applications can add system-dependent information to the object file without causing access utilities to become obsolete.
- Space is provided for symbolic information used by debuggers and other applications.
- Programmers can modify the way the object file is constructed by providing directives at compile time.

The object file supports user-defined sections and contains extensive information for symbolic software testing. An object file contains the following items:

- A file header
- Optional header information
- A table of section headers
- Data corresponding to the section headers
- Relocation information
- Line numbers
- A symbol table
- A string table

Figure 6-1 shows the overall structure.

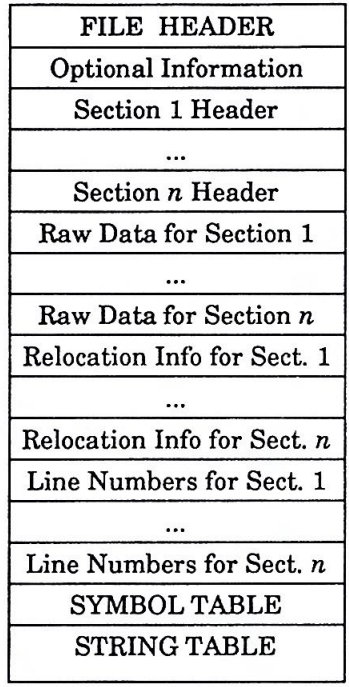


Figure 6-1. Object File Format.

The last four sections (relocation, line numbers, symbol table, and the string table) may be missing if the program is linked with the **–s** option of the **ld** command, or if the line number information, symbol table, and string table are removed by the **strip** command. The line number information does not appear unless the program is compiled with the **–g** option of the **cc** command. Also, if there are no unresolved external references after linking, the relocation information is no longer needed and is absent. The string table is also absent if the source file does not contain any symbols with names longer than eight characters.

An object file that contains no errors or unresolved references is considered executable.

6.2 Definitions and Conventions

Before proceeding further, you should become familiar with the following terms and conventions.

6.2.1 Sections

A section is the smallest portion of an object file that is relocated and treated as one separate and distinct entity. In the most common case, there are three sections named **.text**, **.data**, and **.bss**. Additional sections accommodate comments, multiple text or data segments, shared data segments, or user-specified sections. However, the operating system loads only **.text**, **.data**, and **.bss** into memory when the file is executed.

NOTE

It is a mistake to assume that every COFF file will have a certain number of sections, or to assume characteristics of sections such as their order, their location in the object file, or the address at which they are to be loaded. This information is available only after the object file has been created. Programs manipulating COFF files should obtain such information from file and section headers in the file.

6.2.2 Physical and Virtual Addresses

The physical address of a section or symbol is the offset of that section or symbol from address zero of the address space. The term *physical address* as used in COFF does not correspond to general usage. The physical address of an object is not necessarily the address at which the object is placed when the process is executed. For example, on a system with paging, the address is located with respect to address zero of virtual memory and the system performs another address translation. The section header contains two address fields: a physical address and a virtual address. In all applicable versions of COFF, the physical address is equivalent to the virtual address.

6.2.3 Target Machine

Compilers and link editors produce executable object files that are intended to be run on a particular computer. In the case of cross-compilers, the compilation and link editing are done on one computer with the intent of creating an object file that can be executed on another computer. The term *target machine* refers to the computer on which the object file is destined to run. In the majority of cases, the target machine is the same as the computer on which the object file is being created.

6.3 File Header

The file header contains the 20 bytes of information shown in Table 6-1. The last two bytes are flags that are used by **ld** and object file utilities.

Table 6-1
File Header Contents

Bytes	Declaration	Name	Description
0-1	unsigned short	f_magic	Magic number
2-3	unsigned short	f_nscns	Number of sections
4-7	long int	f_timdat	Time and date stamp indicating when the file was created, expressed as the number of elapsed seconds since 00:00:00 GMT, January 1, 1970
8-11	long int	f_symptr	File pointer containing the starting address of the symbol table
12-15	long int	f_nsyms	Number of entries in the symbol table
16-17	unsigned short	f_opthdr	Number of bytes in the optional header
18-19	unsigned short	f_flags	Flags (see Table 6-2)

6.3.1 Magic Numbers

The magic number specifies the target machine on which the object file is executable.

6.3.2 Flags

The last two bytes of the file header are flags that describe the type of the object file. Currently defined flags are found in the header file **filehdr.h**, and are shown in Table 6-2.

Table 6-2
File Header Flags

Mnemonic	Flag	Meaning
F_RELFLG	00001	Relocation information stripped from the file
F_EXEC	00002	File is executable (i.e., no unresolved external references)
F_LNNO	00004	Line numbers stripped from the file
F_LSYMS	00010	Local symbols stripped from the file
F_AR32W	0001000	32-bit word

6.3.3 File Header Declaration

The C structure declaration for the file header is given below. This declaration may be found in the header file **filehdr.h**.

```
struct filehdr
{
 unsigned short    f_magic;    /* magic number */
 unsigned short    f_nscns;    /* number of section */

 long              f_timdat;   /* time and date stamp */

 long              f_symptr;   /* file ptr to symbol table */
```

```
 long              f_nsyms;    /* number entries in the
                                  symbol table */

 unsigned short    f_opthdr;   /* size of optional header */

 unsigned short    f_flags;    /* flags */
};

#define FILHDR struct filehdr
#define FILHSZ sizeof(FILHDR)
```

6.4 Optional Header Information

The template for optional information varies among different systems that use COFF. Applications place all system-dependent information into this record. This allows different operating systems access to information that only that operating system uses without forcing all COFF files to save space for that information. General utility programs (for example, the symbol table access library functions, the disassembler, and so on) are made to work properly on any common object file. This is done by seeking past this record using the size of optional header information in the file header field **f_opthdr**.

6.4.1 The Standard System a.out Header

By default, files produced by the link editor always have a standard operating system **a.out** header in the optional header field. The system **a.out** header is 28 bytes long. The fields of the optional header are described in Table 6-3.

Table 6-3
Optional Header Contents

Bytes	Declaration	Name	Description
0-1	**short**	**magic**	Magic number
2-3	**short**	**vstamp**	Version stamp
4-7	**long int**	**tsize**	Size of text in bytes
8-11	**long int**	**dsize**	Size of initialized data in bytes
12-15	**long int**	**bsize**	Size of uninitialized data in bytes
16-19	**long int**	**entry**	Entry point
20-23	**long int**	**text_start**	Base address of text
24-27	**long int**	**data_start**	Base address of data

The magic number in the file header specifies the machine on which the object file runs, and the magic number in the optional header supplies information telling the operating system on that machine how that file should be executed. The magic numbers recognized by the operating system are given in Table 6-4.

Table 6-4
System Magic Numbers

Value	Meaning
0407	The text segment is not write-protected or shareable; the data segment is contiguous with the text segment.
0410	The data segment starts at the next segment following the text segment and the text segment is write-protected.
0413	Text and data segments are aligned within **a.out** so it can be directly paged.

6.4.2 Optional Header Declaration

The C language structure declaration currently used for the system **a.out** file header is given in below. This declaration may be found in the header file **aouthdr.h**.

```
typedef struct aouthdr
{
   short     magic;          /* magic number */
   short     vstamp;         /* version stamp */
   long      tsize;          /* text size in bytes, padded */

                             /* to full word boundary */

   long      dsize;          /* initialized data size */

   long      bsize;          /* uninitialized data size */

   long      entry;          /* entry point */

   long      text_start;     /* base of text for this file */

   long      data_start      /* base of data for this file */

} AOUTHDR;
```

6.5 Section Headers

Every object file has a table of section headers to specify the layout of data within the file. The section header table consists of one entry for every section in the file. The information in the section header is described in Table 6-5.

Table 6-5
Section Header Contents

Bytes	Declaration	Name	Description
0-7	char	s_name	8-character null padded section name
8-11	long int	s_paddr	Physical address of section
12-15	long int	s_vaddr	Virtual address of section
16-19	long int	s_size	Section size in bytes
20-23	long int	s_scnptr	File pointer to raw data
24-27	long int	s_relptr	File pointer to relocation entries
28-31	long int	s_lnnoptr	File pointer to line number entries
32-33	unsigned short	s_nreloc	Number of relocation entries
34-35	unsigned short	s_nlnno	Number of line number entries
36-39	long int	s_flags	Flags (see Table 6-6)

The size of a section is padded to a multiple of 4 bytes. File pointers are byte offsets that can be used to locate the start of data, relocation, or line number entries for the section. They can be readily used with the system function **fseek**(3S).

6.5.1 Flags

The lower two bytes of the flag field indicate a section type. The flags are described in Table 6-6.

Table 6-6
Section Header Flags

Mnemonic	Flag	Meaning
STYP_REG	0x00	Regular section (allocated, relocated, loaded)
STYP_DSECT	0x01	Dummy section (not allocated, relocated, not loaded)
STYP_NOLOAD	0x02	Noload section (allocated, relocated, not loaded)
STYP_GROUP	0x04	Grouped section (formed from input sections)
STYP_PAD	0x08	Padding section (not allocated, not relocated, loaded)
STYP_COPY	0x10	Copy section (for a decision function used in updating fields; not allocated, not relocated, loaded, relocation and line number entries processed normally)
STYP_TEXT	0x20	Section contains executable text
STYP_DATA	0x40	Section contains initialized data
STYP_BSS	0x80	Section contains only uninitialized data
STYP_INFO	0x200	Comment section (not allocated, not relocated, not loaded)
STYP_OVER	0x400	Overlay section (relocated, not allocated, not loaded)
STYP_LIB	0x800	For **.lib** section (treated like STYP_INFO)
STYP_SHARED	0x8000	Shared memory section

6.5.2 Section Header Declaration

The C structure declaration for the section headers is described below. This declaration may be found in the header file **scnhdr.h**.

```
struct scnhdr
{
  char      s_name[8];        /* section name */
  long      s_paddr;          /* physical address */
  long      s_vaddr;          /* virtual address */
  long      s_size;           /* section size */
  long      s_scnptr;         /* file ptr to section raw data */

  long      s_relptr;         /* file ptr to relocation */

  long      s_lnnoptr;        /* file ptr to line number */

  unsigned short  s_nreloc;   /* number of relocation entries */

  unsigned short  s_nlnno;    /* number of line number entries */

  long      s_flags;          /* flags */

};

#define  SCNHDR  struct scnhdr
#define  SCNHSZ  sizeof(SCNHDR)
```

6.5.3 .bss Section Header

The one deviation from the normal rule in the section header table is the entry for uninitialized data in a **.bss** section. A **.bss** section has a size and symbols that refer to it, and symbols that are defined in it. At the same time, a **.bss** section has no relocation entries, no line number entries, and no data. Therefore, a **.bss** section has an entry in the section header table but occupies no space elsewhere in the file. In this case, the number of relocation and line number entries, as well as all file pointers in a **.bss** section header, are 0. The same is true of the STYP_NOLOAD and STYP_DSECT sections.

6.6 Sections

Figure 6-1 shows that section headers are followed by the appropriate number of bytes of text or data. The raw data for each section begins on a 4-byte boundary in the file.

Link editor SECTIONS directives allow you to perform the following tasks:

- Describe how input sections are to be combined.
- Direct the placement of output sections.
- Rename output sections.

If no SECTIONS directives are given, each input section appears in an output section of the same name. For example, if a number of object files, each with a **.text** section, are linked together, the output object file contains a single **.text** section made up of the combined input **.text** sections.

6.7 Relocation Information

Object files have one relocation entry for each relocatable reference in the text or data. The relocation information consists of entries with the format described in Table 6-7.

Table 6-7
Relocation Section Contents

Bytes	Declaration	Name	Description
0-3	**long int**	**r_vaddr**	(Virtual) address of reference
4-7	**long int**	**r_symndx**	Symbol table index
8-9	**unsigned short**	**r_type**	Relocation type

The first four bytes of the entry are the virtual address of the text or data to which this entry applies. The next field is the index, counted from 0, of the symbol table entry that is being referenced. The type field indicates the type of relocation to be applied.

As the link editor reads each input section and performs relocation, the relocation entries are read. They direct how references found within the input section are treated. The currently recognized relocation types are given in Table 6-8.

Table 6-8
Relocation Types

Mnemonic	Flag	Meaning
R_ABS	0	Reference is absolute; no relocation is necessary. The entry will be ignored.
R_DIR32	06	Direct 32-bit reference to the symbol's virtual address.
R_DIR32S	012	Direct 32-bit reference to the symbol's virtual address, with the 32-bit value stored in the reverse order in the object file.

6.7.1 Relocation Entry Declaration

The structure declaration for relocation entries is given below. This declaration may be found in the header file **reloc.h**.

```
struct reloc
{
  long           r_vaddr;  /* virtual address of reference */
  long           r_symndx; /* index into symbol table */
  unsigned short r_type;   /* relocation type */
};

#define RELOC    struct reloc

#define RELSZ    10
```

6.8 Line Numbers

When invoked with the **-g** option, the **cc** and **fortran** commands cause an entry in the object file for every source line where a breakpoint can be inserted. You can then reference line numbers when using a software debugger like Pdbx. All line numbers in a section are grouped by function as shown in Table 6-9.

Table 6-9
Line Number Grouping

Function	Line Number
symbol index	0
physical address	line number
physical address	line number
.	.
.	.
.	.
symbol index	0
physical address	line number
physical address	line number

The first entry in a function grouping has line number 0 and has, in place of the physical address, an index into the symbol table for the entry containing the function name. Subsequent entries have actual line numbers and addresses of the text corresponding to the line numbers. The line number entries are relative to the beginning of the function and appear in increasing order of address.

6.8.1 Line Number Declaration

The structure declaration currently used for line number entries is as follows:

```
struct lineno
{
   union
   {
         long   l_symndx;    /* symtbl index of func name */

         long   l_paddr;     /* paddr of line number */
   } l_addr;
   unsigned short   l_lnno;      /* line number */

};

#define LINENO       struct lineno
#define LINESZ       6
```

6.9 Symbol Table

Because of symbolic debugging requirements, the order of symbols in the symbol table is very important. Symbols appear in the sequence shown in Figure 6-2.

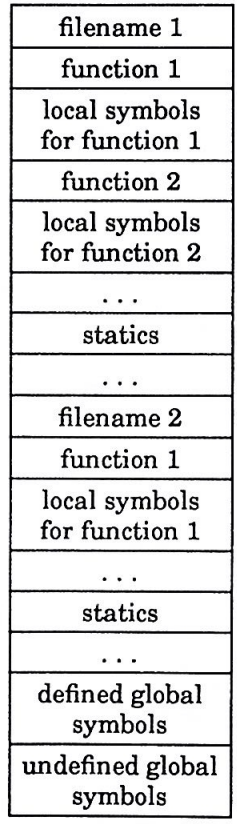


Figure 6-2. COFF Symbol Table.

The word *statics* in Figure 6-2 means symbols defined with the C language storage classify *static* outside any function. The symbol table consists of at least one fixed-length entry per symbol with some symbols followed by auxiliary entries of the same size. The entry for each symbol is a structure that holds the value, the type, and other information.

6.9.1 Special Symbols

The symbol table contains some special symbols that are generated by **as**, and other tools. These symbols are given in Table 6-10.

Table 6-10
Special Symbols in the Symbol Table

Symbol	Meaning
.file	Filename
.text	Address of **.text** section
.data	Address of **.data** section
.bss	Address of **.bss** section
.shdata	Address of **.shdata** section
.shbss	Address of **.shbss** section
.bb	Address of start of inner block
.eb	Address of end of inner block
.bf	Address of start of function
.ef	Address of end of function
.target	Pointer to the structure or union returned by a function
.*x*fake	Dummy tag name for structure, union, or enumeration
.eos	End of members of structure, union, or enumeration
etext	Next available address after the end of the output section **.text**
edata	Next available address after the end of the output section **.data**
end	Next available address after the end of the output section **.bss**
sshadata	First (start) available address for output section **.shdata**
eshdata	Like **edata**
eshbss	Like **end**

Six of these special symbols occur in pairs. The **.bb** and **.eb** symbols indicate the boundaries of inner blocks; a **.bf** and **.ef** pair brackets each function. An **.*x*fake** and **.eos** pair names and defines the limit of structures, unions, and enumerations that were not named. The **.eos** symbol also appears after

named structures, unions, and enumerations.

When a structure, union, or enumeration has no tag name, the compiler invents a name to be used in the symbol table. The name chosen for the symbol table is **.***x***fake**, where *x* is an integer. If there are three unnamed structures, unions, or enumerations in the source, their tag names are **.0fake**, **.1fake**, and **.2fake**. Each of the special symbols has different information stored in the symbol table entry as well as the auxiliary entries.

6.9.2 Inner Blocks

The C language defines a block as a compound statement that begins and ends with braces: { and }. An inner block is a block that occurs within a function (which is also a block).

For each inner block that has local symbols defined, a special symbol **.bb** is put in the symbol table immediately before the first local symbol of that block. Also, a special symbol **.eb** is put in the symbol table immediately after the last local symbol of that block. The sequence is shown in Figure 6-3.

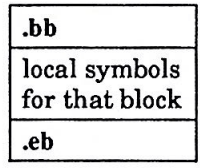


Figure 6-3. Special Symbols (.bb and .eb).

Because inner blocks can be nested by several levels, the **.bb**-**.eb** pairs and associated symbols may also be nested:

```
{                                   /* block 1 */
        int i;
        char c;
        ...
        {                                /* block 2 */
                long a;
                ...
            {                                /* block 3 */
                        int x;
```

```
                    ....
            }                           /* block 3 */

        }                           /* block 2 */

        {                           /* block 4 */
                long i;
                ...
        }                           /* block 4 */
}                           /* block 1 */
```

The symbol table would look like Figure 6-4.

.bb for block 1
i
c
.bb for block 2
a
.bb for block 3
x
.eb for block 3
.eb for block 2
.bb for block 4
i
.eb for block 4
.eb for block 1

Figure 6-4. Example of the Symbol Table.

6.9.3 Symbols and Functions

For each function, a special symbol **.bf** is put between the function name and the first local symbol of the function in the symbol table. Also, a special symbol **.ef** is put immediately after the last local symbol of the function in the symbol table. The sequence is shown in Figure 6-5.

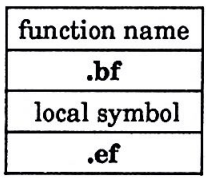


Figure 6-5. Symbols for Functions.

6.9.4 Symbol Table Entries

All symbols, regardless of storage class and type, have the same format for their entries in the symbol table. The symbol table entries each contain 18 bytes of information. The meaning of each of the fields in the symbol table entry is described in Table 6-11. It should be noted that indices for symbol table entries begin at 0 and count upward. Each auxiliary entry also counts as one symbol.

Table 6-11
Symbol Table Entry Format

Bytes	Declaration	Name	Description
0-7	(see text)	_n	These 8 bytes contain either a symbol name or an index to a symbol
8-11	long int	n_value	Symbol value; storage class dependent
12-13	short	n_scnum	Section number of symbol
14-15	unsigned short	n_type	Basic and derived type specification
16	char	n_sclass	Storage class of symbol
17	char	n_numaux	Number of auxiliary entries

Symbol Names

The first eight bytes in the symbol table entry are a union of a character array and two long integers. If the symbol name is eight characters or less, the (null-padded) symbol name is stored there. If the symbol name is longer than eight characters, then the entire symbol name is stored in the string table. In this case, the eight bytes contain two long integers, the first is zero, and the second is the offset (relative to the beginning of the string table) of the name in the string table. Since there can be no symbols with a null name, the zeroes on the first four bytes serve to distinguish a symbol table entry with an offset from one with a name in the first eight bytes as shown in Table 6-12.

Table 6-12
Name Field

Bytes	Declaration	Name	Description
0-7	char	n_name	8-character null-padded symbol name
0-3	long	n_zeroes	Zero in this field indicates the name is in the string table
4-7	long	n_offset	Offset of the name in the string table

Special symbols generated by the C Compilation System are discussed earlier in "Special Symbols."

Storage Classes

The storage class field has one of the values described in Table 6-13. These **#define** statements may be found in the header file **storclass.h**.

Table 6-13
Storage Classes

Mnemonic	Value	Storage Class
C_EFCN	–1	Physical end of a function
C_NULL	0	–
C_AUTO	1	Automatic variable
C_EXT	2	External symbol
C_STAT	3	Static
C_REG	4	Register variable
C_EXTDEF	5	External definition
C_LABEL	6	Label
C_ULABEL	7	Undefined label
C_MOS	8	Member of structure
C_ARG	9	Function argument
C_STRTAG	10	Structure tag
C_MOU	11	Member of union
C_UNTAG	12	Union tag
C_TPDEF	13	Type definition
C_USTATIC	14	Uninitialized static
C_ENTAG	15	Enumeration tag
C_MOE	16	Member of enumeration
C_REGPARM	17	Register parameter
C_FIELD	18	Bit field
C_BLOCK	100	Beginning and end of block
C_FCN	101	Beginning and end of function
C_EOS	102	End of structure
C_FILE	103	Filename
C_LINE	104	Used only by utility programs
C_ALIAS	105	Duplicated tag
C_HIDDEN	106	Like static, used to avoid name conflicts

All of these storage classes except for C_ALIAS and C_HIDDEN are generated by the **cc** or **as** commands. The compress utility, **cprs**, generates the C_ALIAS mnemonic. This utility (described in the **cprs** man page) removes duplicated structure, union, and enumeration definitions and puts alias entries in their places. The storage class C_HIDDEN is not used by any system tools.

Some of these storage classes are used only internally by the C Compilation Systems. These storage classes are C_EFCN, C_EXTDEF, C_ULABEL, C_USTATIC, and C_LINE.

Storage Classes for Special Symbols

Some special symbols are restricted to certain storage classes. They are given in Table 6-14.

Table 6-14
Storage Class by Special Symbols

Special Symbol	Storage Class
.file	C_FILE
.bb	C_BLOCK
.eb	C_BLOCK
.bf	C_FCN
.ef	C_FCN
.target	C_AUTO
.*x*fake	C_STRTAG, C_UNTAG, C_ENTAG
.eos	C_EOS
.text	C_STAT
.data	C_STAT
.bss	C_STAT

Also some storage classes are used only for certain special symbols. They are summarized in Table 6-15.

Table 6-15
Restricted Storage Classes

Storage Class	Special Symbol
C_BLOCK	**.bb, .eb**
C_FCN	**.bf, .ef**
C_EOS	**.eos**
C_FILE	**.file**

Symbol Value Field

The meaning of the value of a symbol depends on its storage class. This relationship is summarized in Table 6-16.

Table 6-16
Storage Class and Value

Storage Class	Meaning of Value
C_AUTO	Stack offset in bytes
C_EXT	Relocatable address
C_STAT	Relocatable address
C_REG	Register number
C_LABEL	Relocatable address
C_MOS	Offset in bytes
C_ARG	Stack offset in bytes
C_STRTAG	0
C_MOU	0
C_UNTAG	0
C_TPDEF	0
C_ENTAG	0
C_MOE	Enumeration value
C_REGPARM	Register number
C_FIELD	Bit displacement
C_BLOCK	Relocatable address
C_FCN	Relocatable address
C_EOS	Size
C_FILE	(see text below)
C_ALIAS	Tag index
C_HIDDEN	Relocatable address

If a symbol has storage class C_FILE, the value of that symbol equals the symbol table entry index of the next **.file** symbol. That is, the **.file** entries form a one-way linked list in the symbol table. If there are no more **.file**

entries in the symbol table, the value of the symbol is the index of the first global symbol.

Relocatable symbols have a value equal to the virtual address of that symbol. When the section is relocated by the link editor, the value of these symbols changes.

Section Number Field

Section numbers are listed in Table 6-17.

Table 6-17
Section Number

Mnemonic	Section Number	Meaning
N_DEBUG	–2	Special symbolic debugging symbol
N_ABS	–1	Absolute symbol
N_UNDEF	0	Undefined external symbol
N_SCNUM	1-077777	Section number where symbol is defined

A special section number (–2) marks symbolic debugging symbols, including structure/union/enumeration tag names, typedefs, and the name of the file. A section number of –1 indicates that the symbol has a value but is not relocatable. Examples of absolute-valued symbols include automatic and register variables, function arguments, and **.eos** symbols.

With one exception, a section number of 0 indicates a relocatable external symbol that is not defined in the current file. The one exception is a multiply defined external symbol (for example, FORTRAN common or an uninitialized variable defined external to a function in C). In the symbol table of each file where the symbol is defined, the section number of the symbol is 0 and the value of the symbol is a positive number giving the size of the symbol. When the files are combined to form an executable object file, the link editor combines all the input symbols of the same name into one symbol with the section number of the **.bss** section. The maximum size of all the input symbols with the same name is used to allocate space for the symbol and the value becomes the address of the symbol. This is the only case where a symbol has a section number of 0 and a non-zero value.

Section Numbers and Storage Classes

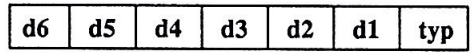
Symbols having certain storage classes are also restricted to certain section numbers. They are summarized in Table 6-18.

Table 6-18
Section Number and Storage Class

Storage Class	Section Number
C_AUTO	N_ABS
C_EXT	N_ABS, N_UNDEF, N_SCNUM
C_STAT	N_SCNUM
C_REG	N_ABS
C_LABEL	N_UNDEF, N_SCNUM
C_MOS	N_ABS
C_ARG	N_ABS
C_STRTAG	N_DEBUG
C_MOU	N_ABS
C_UNTAG	N_DEBUG
C_TPDEF	N_DEBUG
C_ENTAG	N_DEBUG
C_MOE	N_ABS
C_REGPARM	N_ABS
C_FIELD	N_ABS
C_BLOCK	N_SCNUM
C_FCN	N_SCNUM
C_EOS	N_ABS
C_FILE	N_DEBUG
C_ALIAS	N_DEBUG

Type Entry

The type field in the symbol table entry contains information about the basic and derived type for the symbol. This information is generated by the C Compilation System only if the **-g** option is used. Each symbol has exactly one basic or fundamental type but can have more than one derived type. The format of the 16-bit type entry is as follows:



Bits 0 through 3, called **typ**, indicate one of the fundamental types given in Table 6-19.

Table 6-19
Fundamental Types

Mnemonic	Value	Type
T_NULL	0	Type not assigned
T_VOID	1	Void
T_CHAR	2	Character
T_SHORT	3	Short integer
T_INT	4	Integer
T_LONG	5	Long integer
T_FLOAT	6	Floating point
T_DOUBLE	7	Double word
T_STRUCT	8	Structure
T_UNION	9	Union
T_ENUM	10	Enumeration
T_MOE	11	Member of enumeration
T_UCHAR	12	Unsigned character
T_USHORT	13	Unsigned short
T_UINT	14	Unsigned integer
T_ULONG	15	Unsigned long

Bits 4 through 15 are arranged as six 2-bit fields marked **d1** through **d6**. These **d** fields represent levels of the derived types given in Table 6-20.

Table 6-20
Derived Types

Mnemonic	Value	Type
DT_NON	0	No derived type
DT_PTR	1	Pointer
DT_FCN	2	Function
DT_ARY	3	Array

The following examples demonstrate the interpretation of the symbol table entry representing type.

```
char *func();
```

Here **func** is the name of a function that returns a pointer to a character. The fundamental type of **func** is 2 (character), the **d1** field is 2 (function), and the **d2** field is 1 (pointer). Therefore, the type word in the symbol table for **func** contains the hexadecimal number 0x62, which is interpreted to mean a function that returns a pointer to a character.

```
short *tabptr[10][25][3];
```

Here **tabptr** is a three-dimensional array of pointers to short integers. The fundamental type of **tabptr** is 3 (short integer); the **d1**, **d2**, and **d3** fields each contains a 3 (array), and the **d4** field is 1 (pointer). Therefore, the type entry in the symbol table contains the hexadecimal number 0x7f3 indicating a three-dimensional array of pointers to short integers.

Type Entries and Storage Classes

Table 6-21 shows the type entries that are legal for each storage class.

Table 6-21
Type Entries by Storage Class

Storage Class	d Entry			typ Entry Basic Type
	Function?	Array?	Pointer?	
C_AUTO	no	yes	yes	Any except T_MOE
C_EXT	yes	yes	yes	Any except T_MOE
C_STAT	yes	yes	yes	Any except T_MOE
C_REG	no	no	yes	Any except T_MOE
C_LABEL	no	no	no	T_NULL
C_MOS	no	yes	yes	Any except T_MOE
C_ARG	yes	no	yes	Any except T_MOE
C_STRTAG	no	no	no	T_STRUCT
C_MOU	no	yes	yes	Any except T_MOE
C_UNTAG	no	no	no	T_UNION
C_TPDEF	no	yes	yes	Any except T_MOE
C_ENTAG	no	no	no	T_ENUM
C_MOE	no	no	no	T_MOE
C_REGPARM	no	no	yes	Any except T_MOE
C_FIELD	no	no	no	T_ENUM, T_UCHAR, T_USHORT, T_UNIT, T_ULONG
C_BLOCK	no	no	no	T_NULL
C_FCN	no	no	no	T_NULL
C_EOS	no	no	no	T_NULL
C_FILE	no	no	no	T_NULL
C_ALIAS	no	no	no	T_STRUCT, T_UNION, T_ENUM

Conditions for the **d** entries apply to **d1** through **d6**, except that it is impossible to have two consecutive derived types of function.

Although function arguments can be declared as arrays, they are changed to pointers by default. Therefore, no function argument can have array as its first derived type.

Structure for Symbol Table Entries

The C language structure declaration for the symbol table entry is given below. This declaration may be found in the header file **syms.h**.

```
struct syment
{
   union
   {
          char              _n_name[SYMN.LeN];     /* symbol name*/
          struct
          {
                   long     _n_zeroes;     /* symbol name */

                   long     _n_offset;     /* location in string table */
          } _n_n;
          char              *_n_nptr[2];  /* allows overlaying */
   } _n;
   unsigned long     n_value;                /* value of symbol */

   short             n_scnum;                /* section number */

   unsigned short    n_type;                 /* type and derived */

   char              n_sclass;               /* storage class */

   char              n_numaux;               /* number of aux entries */
};

#define  n_name          _n._n_name
#define  n_zeroes        _n._n_n._n_zeroes
#define  n_offset        _n._n_n._n_offset
#define  n_nptr          _n._n_nptr[1]

#define  SYMNMLEN  8
#define  SYMESZ    18    /* size of a symbol table entry */
```

6.9.5 Auxiliary Table Entries

An auxiliary table entry of a symbol contains the same number of bytes as the symbol table entry. However, unlike symbol table entries, the format of an auxiliary table entry of a symbol depends on its type and storage class. They are summarized in Table 6-22.

Table 6-22
Auxiliary Symbol Table Entries

Name	Storage Class	Type Entry		Auxiliary Entry Format
		d1	typ	
.file	C_F.Le	DT_NON	T_NULL	Filename
.text,.data, .bss	C_STAT	DT_NON	T_NULL	Section
tagname	C_STRTAG C_UNTAG C_ENTAG	DT_NON	T_NULL	Tag name
.eos	C_EOS	DT_NON	T_NULL	End of structure
fcname	C_EXT C_STAT	DT_FCN	(Note 1)	Function
arrname	(Note 2)	DT_ARY	(Note 1)	Array
.bb,.eb	C_BLOCK	DT_NON	T_NULL	Beginning and end of block
.bf,.ef	C_FCN	DT_NON	T_NULL	Beginning and end of function
Name related to structure, union, enumeration	(Note 2)	DT_PTR, DT_ARR, DT_NON	T_STRUCT, T_UNION, T_ENUM	Name related to structure, union, enumeration

Note 1: Any except T_MOE
Note 2: C_AUTO, C_STAT, C_MOS, C_MOU, C_TPDEF

In Table 6-22, *tagname* means any symbol name including the special symbol **.xfake**, and *fcname* and *arrname* represent any symbol name for a function or an array respectively. Any symbol that satisfies more than one condition in Table 6-22 should have a union format in its auxiliary entry.

NOTE

It is a mistake to assume how many auxiliary entries are associated with any given symbol table entry. This information is available, and should be obtained from the **n_numaux** field in the symbol table.

Filenames

Each of the auxiliary table entries for a filename contains a 14-character filename in bytes 0 through 13. The remaining bytes are 0.

Sections

The auxiliary table entries for sections have the format as shown in Table 6-23.

Table 6-23
Format for Auxiliary Table Entries for Sections

Bytes	Declaration	Name	Description
0-3	**long int**	**x_scnlen**	Section length
4-5	**unsigned short**	**x_nreloc**	Number of relocation entries
6-7	**unsigned short**	**x_nlinno**	Number of line numbers
8-17	–	–	Unused (filled with zeros)

Tag Names

The auxiliary table entries for tag names have the format shown in Table 6-24.

Table 6-24
Tag Names Table Entries

Bytes	Declaration	Name	Description
0-5	–	–	Unused (filled with zeros)
6-7	unsigned short	x_size	Size of structure, union, and enumeration
8-11	–	–	Unused (filled with zeros)
12-15	long int	x_endndx	Index of next entry beyond this structure, union, or enumeration
16-17	–	–	Unused (filled with zeros)

End of Structures

The auxiliary table entries for the end of structures have the format shown in Table 6-25.

Table 6-25
Table Entries for End of Structures

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	Tag index
4-5	–	–	Unused (filled with zeros)
6-7	unsigned short	x_size	Size of structure, union, or enumeration
8-17	–	–	Unused (filled with zeros)

Functions

The auxiliary table entries for functions have the format shown in Table 6-26.

Table 6-26
Table Entries for Functions

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	Tag index
4-7	long int	x_fsize	Size of function (in bytes)
8-11	long int	x_lnnoptr	File pointer to line number
12-15	long int	x_endndx	Index of next entry beyond this point
16-17	unsigned short	x_tvndx	Index of the function's address in the transfer vector table

Arrays

The auxiliary table entries for arrays have the format shown in Table 6-27. Defining arrays having more than four dimensions produces a warning message.

Table 6-27
Table Entries for Arrays

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	Tag index
4-5	unsigned short	x_lnno	Line number of declaration
6-7	unsigned short	x_size	Size of array
8-9	unsigned short	x_dimen[0]	First dimension
10-11	unsigned short	x_dimen[1]	Second dimension
12-13	unsigned short	x_dimen[2]	Third dimension
14-15	unsigned short	x_dimen[3]	Fourth dimension
16-17	–	–	Unused (filled with zeros)

End of Blocks and Functions

The auxiliary table entries for the end of blocks and functions have the format shown in Table 6-28.

Table 6-28
End of Block and Function Entries

Bytes	Declaration	Name	Description
0-3	–	–	Unused (filled with zeros)
4-5	unsigned short	x_lnno	C-source line number
6-17	–	–	Unused (filled with zeros)

Beginning of Blocks and Functions

The auxiliary table entries for the beginning of blocks and functions have the format shown in Table 6-29.

Table 6-29
Format for Beginning of Block and Function

Bytes	Declaration	Name	Description
0-3	–	–	Unused (filled with zeroes)
4-5	unsigned short	x_lnno	C-source line number
6-11	–	–	Unused (filled with zeroes)
12-15	long int	x_endndx	Index of next entry past this block
16-17	–	–	Unused (filled with zeroes)

Names Related to Structures, Unions, and Enumerations. The auxiliary table entries for structure, union, and enumeration symbols have the format shown in Table 6-30.

Table 6-30
Entries for Structures, Unions, and Enumerations

Bytes	Declaration	Name	Description
0-3	**long int**	**x_tagndx**	Tag index
4-5	–	–	Unused (filled with zeros)
6-7	**unsigned short**	**x_size**	Size of the structure, union, or enumeration
8-17	–	–	Unused (filled with zeros)

Aggregates defined by **typedef** may or may not have auxiliary table entries. For example:

```
typedef struct people STUDENT;

struct people
{
        char name[20];
        long id;
};

typedef struct people EMPLOYEE;
```

The symbol EMPLOYEE has an auxiliary table entry in the symbol table, but symbol STUDENT will not because it is a forward reference to a structure.

Auxiliary Entry Declaration

The C language structure declaration for an auxiliary symbol table entry is given below. This declaration may be found in the header file **syms.h**.

```
union auxent
{
      struct
      {
            long   x_tagndx;
            union
```

```
        {
            struct
            {
                  unsigned short   x_lnno;
                  unsigned short   x_size;
            } x_lnsz;
            long    x_fsize;
        } x_misc;
        union
        {
            struct


.
.
.
            {
                  long    x_lnnoptr;
                  long    x_endndx;
            } x_fcn;
            struct
            {
                  unsigned short   x_dimen[DIMNUM];
            } x_ary;
        } x_fcnary;
        unsigned short   x_tvndx;
    } x_sym;
    struct
    {
        char   x_fname[FILN.LeN];
    } x_file;
    struct
    {
        long   x_scnlen;
        unsigned short   x_nreloc;
        unsigned short   x_nlinno;
    } x_scn;
    struct
    {
        long   x_tvfill;
        unsigned short   x_tvlen;
        unsigned short   x_tvran[2];
    } x_tv;
```

```
}
#define FILNLIN  14
#define DIMNUM    4
#define AUXENT    union auxent
#define AUXESZ    18
```

6.10 String Table

Symbol table names longer than eight characters are stored contiguously in the string table with each symbol name delimited by a null byte. The first four bytes of the string table are the size of the string table in bytes. This means that offsets into the string table are greater than or equal to 4. For example, given a file containing two symbols (with names longer then eight characters, **long_name_1** and **another_one**) the string table has the format shown in Figure 6-5.

'l'	'o'	'n'	'g'
'_'	'n'	'a'	'm'
'e'	'_'	'l'	'\0'
'a'	'n'	'o'	't'
'h'	'e'	'r'	'_'
'o'	'n'	'e'	'\0'

Figure 6-5. String Table.

The index of **long_name_1** in the string table is 4 and the index of **another_one** is 16.

6.11 Access Routines

The operating system contains a set of access routines that are used for reading the various parts of a common object file. Although the calling program must know the detailed structure of the parts of the object file it processes, the routines effectively insulate the calling program from the knowledge of the overall structure of the object file.

The access routines can be divided into four categories:

1. Functions that open or close an object file
2. Functions that read header or symbol table information
3. Functions that position an object file at the start of a particular section of the object file
4. A function that returns the symbol table index for a particular symbol

These routines can be found in the library **libld.a** and are listed in Section 3 of the man pages. A summary of what is available can be found in the **ldfcn**(4) man page.

Index

.profile 5-5,14

A

a.out 1-8
A_COLOR 5-46
addch 5-19,20,22,23,37,54,71
addstr 5-12,13,19,22,23,36
advisory 3-1
alcurses 5-3
archive ..... 1-9,28,37,57 to 62; 2-18,23,28 to 29
argument ..... 1-11 to 13,23
  mode 1-30
assembler 1-6
assembly language 1-4,6
attroff 5-39
attron 5-39,44
attrset 5-37,38,39,44,93
awk ..... 1-4 to 5; 2-4 to 5

B

bc 1-6
beep 5-50
bessel 2-25
bitmasks 5-38
box 5-66
BullsEye 5-3,5

C

C Language 1-2
  data types 1-2
  features 1-2
can_change_colors 5-46
captoinfo 5-7,8,85
cbreak 5-51,55,96
cc ..... 1-6 to 8; 2-30,43

cflow 1-43,44
child. *See* fork
chtype 5-20,37
clear 5-19,27
clear_screen 5-71
clearok 5-27
clrtobot 5-19,28,96
clrtoeol 5-19,28,29,96
COBOL 1-4
COFF ..... 1-6; 2-18 to 19; Chapter 6
color, HSL notation 5-41
color, initializing 5-42
color, RGB notation 5-41,46
color-pair 5-41
color-pair, initializing 5-44
color-pairs table 5-43
color_content 5-46
COLOR_PAIR 5-44
colors ..... table
colors ..... program
colors table, default 5-42
colors table, redefining 5-44
See also COFF
compiling. *See* cc
ctrace 1-44,47
curses 1-28
  introduction to 1-6
curses ..... 5-1 to 3; 5-5 to 12; 5-14,18,19,29,52,65,67,68,74,87
  library 5-4
curses, program examples ..... 5-86 to 103
curses.h 5-3,10,11,12,14,20,71
See also header
cxref ..... 1-47 to 52

D

dc 1-6
deadlock 3-13,15
documentation 2-2
doupdate 5-11,57,58,67,68

E

echo 5-54
editor 5-86,87
endwin 5-3,10,11,13,25
environment ..... 1-13,32 to 39; 2-30
erase 5-19,27
error. *See* ctrace
exclusive 3-2
exec ..... 1-13,34 to 35,36

F

fcntl ..... 2-12 to 13; 3-8 to 9
fgets 1-15,22,23,50
fildes 2-13
flash 5-50
fork ..... 1-34,35 to 36; 2-14
FORTRAN 1-3

G

getc 5-2
getch 5-2,29,31,32,33,56,71
getchar 5-29
getstr 5-29,33,35,56
getting message queues 4-8
graphics 5-65

H

halfdelay 5-53
has_colors 5-46
header 1-6,23,27,28; 2-7,10,12,14,18,20,23
header file, **curses.h** 5-10
header files 5-70
highlight 5-93

I

I/O ..... 1-29 to 31; 5-37,56,65
idlok 5-96

inch 5-38
infocmp 5-7,8,84
init 5-9
init_color 5-42,45,49
init_pair 5-43,44,48
initscr 5-3,5,10,11,12,25,27,67
insert_line 5-74
Interprocess Communication ..... Chapter 4
IPC ..... 2-14 to 16; Chapter 4

J-K

key and id ..... 4-1 to 3
keyboard-entered capabilities 5-81
keypad 5-29

L

labels 5-66,67
labels, soft 5-66
languages 2-3,10
lcurses 5-13
ld 1-8; 2-11
ld. *See* link editor
lex ..... 1-5,43; 2-4,5 to 8
libc 1-9,14,28,58; 2-19,28
library 3-1
library, **curses** 5-1
link editor ..... 1-6,8 to 9; 2-8,17 to 18
lint 1-25,50,52; 2-1,17,29
lockf ..... 2-13 to 14; 3-1,15
locking. *See* lockf
longname 5-9

M

M4 1-5
macro 1-5,19,20,28,29,31,57; 2-23
make ..... 1-56 to 57; 2-30 to 31
malloc 2-10

man pages
location of 1-21
mathematical 1-3,6
memory management 2-10
message ..... 4-3 to 29
move 5-12,13,19,25
msgctl, using 4-14
msgget control commands 4-10
msgget, using 4-8
msgop, using 4-20
mvaddch 5-19,25
mvaddstr 5-19,25
mvinch 5-86
mvprintw 5-19

N

newwin 5-61,62
nobreak 5-51
nocbreak 5-54,55
nodelay 5-53
noecho 5-51,54,96
noraw 5-53

O

object 2-18,19
object. *See* COFF
object file libraries 1-28

P

pad 5-18
pads 5-18,56,57
pair_content 5-46
PAIR_NUMBER 5-46
parameter string capabilities 5-81
parent. *See* fork
parsing 2-4,6,10
Pascal 1-3
pdbx ..... 1-6,7,55 to 56
Pdbx 2-29

semop, using 4-52
set_term 5-69
setupterm 5-5,71
shared memory data structure 4-58
shared memory identifier 4-58
shared memory operation permissions codes 4-63
shared memory segment 4-58
shared memory state information 4-60
shared memory, using 4-58
shell 1-13,32,33,58; 2-6
See also pipes
shmctl, using 4-68
shmget control commands 4-64
shmget, using 4-62
shmop, using 4-74
single-process program 5-68
size 1-55
slk_attroff 5-68
slk_attron 5-68
slk_attrset 5-68
slk_clear 5-68
slk_init 5-67
slk_int 5-67
slk_noutrefresh 5-67
slk_refresh 5-67
slk_set 5-67
standend 5-38,40
standout 5-38,40
start_color 5-10,44,47
stderr 1-29,30
stdio 1-23
See also header
stdio 5-19,29
See also header
stdout 5-2
stdscr 5-2,14,15,17,19,20,22,23,25,27,37,56,67,86,101
stream 1-23
string operations 1-18
strip 1-55
subroutines ..... 1-13 to 26
  use of ..... 1-21 to 26
subwin 5-61,63

supported languages 1-2
system calls ..... 1-13 to 26
  use of ..... 1-21 to 26

T

TERM 5-5,8,84
term.h 5-71
termcap 5-8,78,85
termhl 5-71,75
terminal capabilities 5-76
terminal capabilities, basic 5-79
terminal capabilities, keyboard-entered 5-81
terminal capabilities, parameter string 5-81
terminal capabilities, screen-oriented 5-80
terminal description 5-75,83
terminfo ..... 2-16 to 17
TERMINFO 5-8,83
terminfo ..... 5-1 to 5; 5-7 to 9; 5-12,70,71,74 to 77;85
terminfo database 5-44
See also header
tic 5-4,7,83
tools
  advanced 2-10
touchwin 5-101
tparm 5-81
tput 5-7,9,74,84
tputs 5-71,74
trigonometric functions 2-25

U-V-W-X-Y-Z

underline_char 5-74
WINDOW 5-14
windows 5-10,17,18,56
wnoutrefresh 5-11,57,58,67
wrefresh 5-11,14,101
yacc 1-5; 2-4,5,8,10